Unifying Representation and Generalization:

Understanding Hierarchically Structured Objects

Kenneth Hal Wasserman

CUCS-177-85

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Science

COLUMBIA UNIVERSITY
1985

This research was supported in part by the Defense Advanced Research Projects Agency under contract N00039-84-C-0165 and by a fellowship from Philips Laboratories.

© 1985
Kenneth Hal Wasserman
ALL RIGHTS RESERVED

ABSTRACT

Hierarchies are pervasive. They are used to organize and describe many artificial and natural phenomena. In general, humans are very good at understanding them. It therefore seems reasonable to give computers the same ability if they are to be "intelligent".

The integration of representation and generalization is necessary in order to understand hierarchically structured objects. In this thesis we address the issues involved and present a scheme, MERGE, designed to be used in computer systems that understand and automatically classify instances of hierarchies in a given domain.

The MERGE scheme uses a form of dynamic generalization-based memory in order to achieve this integration. Representations of individual hierarchies are stored in terms of how they vary from previously created generalized concepts. Memory is continually reorganized as new data becomes available to a MERGE-based system so that it accurately reflects the known information. The overall effect of this scheme is that representations of individual hierarchies are enhanced by the use of information in the knowledge base. These representations are in turn used to enhance the knowledge base by permitting more and better generalizations to be made.

We have developed two MERGE-based computer systems that intelligently understand hierarchies. CORPORATE-RESEARCHER is a program that learns about upper-level corporate management hierarchies when it is fed representations of corporate charts. RESEARCHER is a larger, natural language processing program that reads and understands patent abstracts about physical objects. Both programs serve as intelligent information systems that automatically classify representations of instance hierarchies.

Table of Contents

List of Figures

Acknowledgments

First and foremost I would like to thank Michael Lebowitz for everything he has done in helping me complete my degree. He has helped originate many of the concepts developed in this work through interesting and insightful discussions. His reading and commenting on earlier versions of this work and others has led to greatly improved content and style in my writings. He has been extremely helpful and encouraging in every respect. This thesis would not have been possible without him.

I would like to thank Kathy Mckeown for providing a balanced view of much of my research over the past several years. Her suggestions and insight have been instrumental in improving this work.

John Kender deserves credit for suggesting the idea of hierarchy understanding in general. He has also reviewed and given useful comments on improving earlier drafts of this work, which is greatly appreciated.

Mike Wish and Jim Corter have been very helpful members of my thesis committee. Their comments on an earlier version of this work have been important in improving its quality.

I owe a great deal to my friends and colleges in the RESEARCHER group at Columbia. Tom Ellman and Cecile Paris have made significant contributions to this thesis. They have been a source of many ideas and good times. Larry Hirsch, Ursula Wolz, and Michelle Baker have also contributed much through long discussions and constructive criticism. I very much appreciate the support and encouragement of all of them.

My friends Jim Kurose, Julie Kurose, Joel Maxman, Elyse Victor, and Mary Ann Stephens have been equally supportive of my work and my sincere thanks goes out to them. Jim deserves special thanks for having endured as my office-mate and sounding board for many years. Mary Ann has helped greatly in the final preparation of this work.

Lastly, and most importantly, my family has been exceptionally wonderful in encouraging my work and tolerating my complaints. To Grace, Arthur, Jeffrey, Winnie, Mona, Jonathan, and Jared thanks hardly expresses my appreciation of their support.

The MERGE scheme of hierarchy understanding is a form of generalization-based memory that integrates representations of individual instances with generalizations created from them. The result is an enhancement of both representation and generalization over schemes that treat each process separately. The need to use MERGE to understand hierarchies comes about because hierarchical systems are often too complex for people to grasp in detail and because there are domains with large numbers of instances that must be understood (generalized). A MERGE-based system can automatically classify large numbers of hierarchies in an incremental fashion while learning about all sub-hierarchies within the whole. An example of MERGE is shown by giving a brief demonstration of CORPORATE-RESEARCHER, a program that understands corporate hierarchies. The originality of this research is clearly indicated, and a preview of the thesis is given.

1. Introduction

Representation and generalization have long been treated as separate problems in Artificial Intelligence (AI). It is our contention that representation and generalization must be unified in order to build intelligent information systems (as in [Lebowitz 83a]). In particular, we will show that a dynamic generalization-based memory (GBM) scheme can be used to understand hierarchically structured objects.

Our scheme is an integration of a representation formalism with generalization techniques for use in understanding hierarchically structured objects (tangible or intangible). When objects are stored in terms of their similarities to, or differences from, other objects or generalizations in memory, a dynamic memory system (in the sense that it changes with the knowledge it stores, see [Schank 82]) can be realized. Such a dynamic system uses generalization as a method to enhance representation, chiefly by creating and changing a classification hierarchy as new data becomes available. Furthermore, this classification hierarchy will dictate the way in which objects analyzed by the system are represented. The overall effect is a unification of representation and generalization into a GBM structure that we call *Mutually Enhanced Representation and GEneralization* (MERGE).

1.1 Overview

The intent in developing the MERGE scheme is to use it as the heart of intelligent information systems that understand hierarchically structured objects. Much of what we perceive in the world around us is hierarchical. Physical objects, river and road systems, library systems, family relations, and all kinds of taxonomies are examples of common hierarchical phenomena. Humans seem adept at understanding information presented as hierarchies. People even create abstract hierarchically structured systems when entertaining themselves; most western music

fits this form. It seems logical that artificial intelligence programs should have a similar capacity to comprehend hierarchies.

An intelligent information system that understands hierarchically structured objects would be useful for two reasons. First, some hierarchies are extremely complex. For example, an automobile can be viewed as a hierarchy of parts that range from the entire car down to the screws that secure a gasket in the water pump of the engine. Being able to simultaneously make comparisons of cars through all these levels of detail is a difficult task for humans. An automated system would be ideal for such a task.

The second reason to use an intelligent information system to understand hierarchies is to categorize instances in domains with many instances. For example, a biological taxonomy classifies animals based upon each animal's structure (i.e., a hierarchy of body parts). A system that can automatically create a classification of animals, based on their structure, could be a help to biological taxonomists. The MERGE scheme offers a way to automatically create such a classification. (An implementation of MERGE for zoological taxonomy would somehow have to capture established classifications, instead of creating its own classes, if it were to be really useful.)

Building an automated system that understands hierarchies is a difficult task. The difficulties arise from having to represent arbitrarily complex hierarchical systems that may or may not be described in a canonical fashion. Furthermore, for the understanding system to be useful for many real-world tasks it has to learn incrementally. That is, it must process data as it becomes available to the system. It can not have the luxury of having all instance hierarchies available for analysis at one time. On top of these difficulties, an automated, intelligent understander of hierarchies must be able to successfully represent individual and generalized hierarchies.

It is our contention that representation and generalization must be integrated in order to achieve a true understanding system. The main reason (in this thesis) for analyzing issues in representation and generalization in isolation is that the complexity of the MERGE scheme makes it difficult to comprehend all at once. Breaking it down into two parts makes a discussion of the scheme's elements more tractable. In addition, this separation facilitates referencing past research, emphasizing where our work departs from it, and showing why an integration of representation and generalization is needed for an understanding system.

After looking at generalization and representation, the details of the MERGE scheme itself will be given. Two computer systems that employ MERGE as the basis of their operation are discussed. CORPORATE-RESEARCHER is a program that understands the information supplied by corporate charts (i.e., it learns about upper-level corporate management structures, as demonstrated later in this chapter).

It is used mostly as a means to demonstrate the features of MERGE. RESEARCHER [Lebowitz 83b] is an intelligent information system that reads, understands, and answers questions about patent abstracts. It learns about complex physical objects, disc drives in particular. Both corporate charts and disc drives are examples of hierarchically structured objects, which is what MERGE is designed to understand.

1.1.1 Definitions

The word ***understand*** has been used to describe the actions of intelligent systems without stating exactly what is meant by this term. When discussing hierarchies, we will limit the meaning of the word "understand" to the recognition of how objects differ from (or are similar to) other objects or generalizations in the same domain and the creation of new generalizations that embody these observations. Specifically, this means structuring memory using generalizations of instance objects and creating further generalizations upon these. In fact, a hierarchy of generalizations (in essence a classification hierarchy) is what the understanding process creates. Each generalization is a comparison made between at least two instance objects (or other generalizations). The result of this comparison is a new memory element that represents the information the objects have in common.

We use the term ***instance*** to refer to an object that is a specific example of something in the real world. When two or more instance objects are compared and a generalization of them is created it is called a ***generalized object***. Generalized objects have instance objects and/or other generalized objects as ***variants***. That is, an instance object is included in the class defined by the generalized object of which it is a variant. Similarly, generalized objects may be included in a class defined by another generalized object by making them variants of this "higher-level" concept.

A ***hierarchy*** describes a system in which each member of the system exists in some partially ordered state relative to the other members. Usually a hierarchy appears as a strict tree structure, with each member (node) of the hierarchy being subservient to exactly one other node. This is the definition that will be used throughout this thesis. However, a broader definition of a hierarchy will occasionally be referred to. A ***tangled hierarchy*** or ***almost-hierarchy*** [Sussman and Steele 80] is one in which each node may be subservient to more than one other node.

Knowledge representation in AI is concerned with formal representation schemes and the processing of knowledge within these schemes. Our work can be considered to fall within the confines of this area. The MERGE scheme is a formal representation scheme as well as a method for processing this information. However, we will use the term ***representation*** specifically to refer to formal systems with which knowledge is encoded, not how it is used.

MERGE uses a fixed knowledge representation formalism (defined in Chapters 3 and 4) that is particularly well suited to hierarchy representation. It uses this language to encode individual instance hierarchies and new generalizations in terms of previously created generalizations. It dynamically modifies the data needed for representation -- it does not modify its knowledge representation language.

Generalization is the type of learning that MERGE carries out. Specifically, MERGE is designed to learn incrementally by modifying a knowledge base to continually reflect the known state of the world. This is opposed to all-at-once learning where information is amassed before it is analyzed. Unless stated otherwise, *learning* will refer to the incremental generalization process.

1.2 The need for hierarchy understanding systems

Hierarchies are pervasive. They are encountered in everyday situations as well as in scientific pursuits. Since a goal of AI is to understand the same realm of information as humans do, considerable attention must be paid to the comprehension of hierarchies. The practical motivations for developing automated hierarchy understanding systems have been mentioned above. The details of real-world hierarchies are often difficult for people to grasp all at one time because of their complexity. They may be able to understand a few levels in a hierarchy or a few lineages, but entire hierarchical systems are often overwhelmingly complex. Additionally, some domains have large numbers of instance hierarchies, too many for any one person to understand.

We first look at several examples of hierarchical domains. Following this, the usefulness of an automated understanding system for hierarchies will be discussed further.

1.2.1 Hierarchical domains

Figure 1-1 shows several examples of hierarchies in various domains. Two major types of hierarchical systems are included (as distinguished by [Simon 81]): ***natural hierarchies***, those that nature has formed and that humans perceive as being hierarchical; and ***artificial hierarchies***, those that are purely human inventions. There are a few hierarchies that are not clearly natural or artificial. For example, atomic structure is actually a model of what people believe to be a hierarchy in nature. (The model is an artificial hierarchy, but it purports to represent a natural hierarchy.) A domain can have examples of hierarchies from either or both of these classes. Physical objects can be man-made (artificial) as are automobiles, disc drives, and so on. Or they can be natural hierarchies, as is the case with trees (e.g., Maple, Oak, etc.) [Rosch et al. 76; Hemenway and Tversky 84].

Some hierarchies are obvious. Governments, military establishments, and corporations make charts describing their tree-like structures. Biological taxonomies

Hierarchical Domain	Examples
Human Organizational Systems	governments(a), corporations(a), clubs(a), religious institutions(a)
Physical Objects	plant and animal physiology(n), atomic and cosmic structure, automobiles(a), buildings(a)
Human Symbolic Systems	writing(a), music(a), library organizational systems(a)
Taxonomies	biological taxonomies, sub-atomic particle classification, chemical classification
Genealogy	family trees(n)
Road/River Systems	highway location planning(a), rivers and their tributaries(n)

Several examples of types of hierarchies in each domain are shown. Both natural and artificial (man-made) hierarchies are given as examples under some domains. Artificial hierarchies are indicated by an "a" in parentheses after each example. Natural hierarchies are shown with an "n" following them. The examples that have an unidentified type are described in the text

Figure 1-1: Hierarchical domains.

and family trees are explicitly made to be hierarchical in form. Most forms of writing are also hierarchical and an author usually tries to keep this in mind while writing long documents (e.g., this thesis is a hierarchy of words, sentences, paragraphs, sections, and chapters).

Other hierarchies are not so obvious. Musicologists describe compositions as being hierarchies of notes, measures, phrases, themes, and higher level structures. The process of deciding how to plan the routing of a highway is structured as a hierarchical decision task [Manheim 66]. Even buildings are hierarchical in structure; they have rooms, suites of rooms, floors, and clusters of floors (in some large office buildings).

Figure 1-2 shows several examples of hierarchies that computer scientists use. The right hand column shows an *organizational concept* for each type of hierarchy. Hierarchical domains can have several ways to organize information. Thus, it is necessary to specify which of these organizational concepts is being studied. We will focus on this point when the ***fundamental relation*** (*F-rel*) of a hierarchy is

discussed in Chapter 3. Simply put, the F-rel is a formal characterization of a hierarchy's organizational concept.

Type of Hierarchy	Organizational Concept
grammatical parse trees	part of a phrase
structured programming	one module calls another
communications protocols	passes data up/down through levels
disc file management	increasing/decreasing units of size
hardware organization	in same subsystem as
circuit layout	functionally part of

Some of the many examples of hierarchies found just in the computer science domain are listed. The *organizational concept* indicates what the basis for each hierarchy is.

Figure 1-2: Hierarchies within computer science.

1.2.2 Automatic classification

The wide variety of hierarchies in the real world makes the need for an understanding of them important. Although humans are able to recognize and construct hierarchies, they are not particularly good at simultaneously understanding all the levels of detail that a single, complex hierarchy can represent.

A hierarchy can be seen as a recursive structure. That is, a hierarchy is composed of a root node and other sub-hierarchies. Each sub-hierarchy being another hierarchy with one less level of detail than its predecessor. Representing this recursive structure can be easily done on a computer. Since computers have accurate memories they can "remember" a lot of detail. Thus, a properly designed program can represent hierarchies of arbitrary depth (level of detail).

Having such a representation scheme is only a prerequisite to understanding hierarchies, not the method. Understanding has been defined to include generalization in this thesis. Consequently, instance hierarchies can only be understood in relation to other hierarchies; they can not be understood in isolation. Some generalization-type processing must be done on representations of hierarchies in order to understand them.

Throughout this thesis two kinds of hierarchies are discussed. Hierarchies that represent a specific instance of a system or a generalization of a system are called

F-trees. For example, a representation of John's 1984 Toyota, Mary's corporation, or a generalized modular computer program is an F-tree. (The origin of the term, F-tree, is described in Chapter 3.) The other type of hierarchy that will be discussed is called a Generalization tree, or *G-tree*. A G-tree is composed of generalized concepts and specific instances. Each node (concept) in a G-tree is an F-tree. Thus, a G-tree is a hierarchy of generalizations of F-trees. The leaves of a G-tree usually are specific F-trees (instance hierarchies). Higher levels in a G-tree are generalized F-trees, with the most general F-tree situated at the root of the G-tree.

Understanding involves creating and maintaining G-trees. For example, an understanding of household furniture would necessitate the creation of concepts of various classes, types, and kinds of furniture. There might be bedroom furniture and living-room classes. Types of living-room furniture would be chairs, couches, and tables. The kind of chair would be the next classification level (e.g., with or without arms). At the lowest level of this hierarchy would be specific pieces of furniture from a specific manufacturer.

When a domain such as household furniture has a large number of instances, an intelligent understanding system also serves as a way to automatically classify objects. That is, the understanding process creates a hierarchy of generalized concepts. These concepts are analogous to the classes, types, and kinds that are needed to classify furniture. In general, a hierarchical categorization of objects can have arbitrarily many levels.

MERGE-based systems are particularly useful for automatic classification of objects in domains with many instances. It seems to use that people are often unable to generalize about large numbers of similar objects and still recall the details of a particular object. To compensate for this problem they resort to external means (e.g., database systems) to assist them in storing knowledge about such domains. However, humans must make the generalizations (i.e., we are assuming that databases, heretofore, cannot make generalizations). Thus, representations of objects are separated from the generalizations about them. Although this separation may allow a human to perform better at classifying objects, it is inflexible in that an addition to the database may require human intervention to reclassify the information. Our scheme allows large numbers of objects to be both represented and generalized about within the same environment, eventually leading to an integrated, automated, intelligent information system.

A MERGE-based system can automatically classify several objects simultaneously. Since the objects being classified by MERGE are hierarchical in structure, several categorization hierarchies can be created during the understanding process -- one categorization hierarchy for each unique sub-part of an object. The MERGE scheme enables the understanding of parts of objects during the process of understanding the whole object. People commonly learn about objects within the

context of other objects. For example, they might learn about random access memory (RAM) and external memory (e.g., disc drives) while trying to understand personal computers. We believe that intelligent information systems should duplicate this type of learning if they are to be robust in their ability to process varied input data.

1.3 MERGE-based systems

The MERGE scheme can be applied to the task of understanding in any domain comprised of hierarchically structured objects. However, it is limited to use in one area at a time (e.g., computer disc drives within the physical object domain). An implementation of MERGE must be focused on a specific area of a domain.

We have built two related programs that employ MERGE as the basis for organizing memory to understand hierarchies. CORPORATE-RESEARCHER is designed to learn about upper-level corporate management hierarchies. As such, the objects it understands are corporate officers, divisions, or departments. RESEARCHER is a system that reads patent abstracts about disc drives, thereby learning about how they are structured. It understands hierarchies that describe complex physical objects.

Aside from the fundamentally different objects that these programs represent and generalize about, they have other differences. RESEARCHER is a large natural language processing system that obtains its data by parsing English language text into MERGE's representational formalism. CORPORATE-RESEARCHER gets its input from hand encodings of corporate charts.

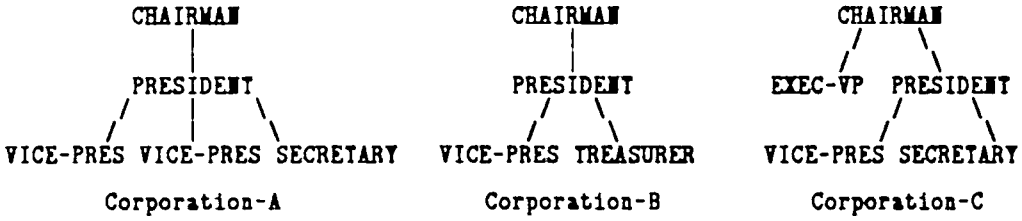
Both systems use information (relations) to supplement the basic hierarchical representations. Although RESEARCHER and CORPORATE-RESEARCHER both make use of relations they do so to different degrees. RESEARCHER has a sophisticated method for representing physical relations among parts of an object hierarchy. For example, a disc drive may have a read/write head ON-TOP-OF a disc (with both being parts of the disc drive). The relations that CORPORATE-RESEARCHER uses have to do with special interactions among various branches of a corporation. For example, an acquisitions committee ADVISES the chairman of the board. Relations are of importance in augmenting a single tree-like representation of a hierarchy so that it more closely captures reality.

The fact that CORPORATE-RESEARCHER and RESEARCHER understand significantly different domains with different input sources and relational information is important. If MERGE can successfully be used in these two programs then it can likely be applied to a wide range of hierarchical domains. We will examine the details of these programs in more depth in Chapter 6, but will look at a simple example of how MERGE works here.

1.3.1 CORPORATE-RESEARCHER

We choose to briefly demonstrate MERGE with an example from CORPORATE-RESEARCHER. (We will not show actual program output, but this example has been processed by CORPORATE-RESEARCHER.) The hierarchies that this program understands are very straightforward in that corporate management structures are generally tree-like in form, as corporate charts show. In addition, the rankings of members of a corporation are known to most people (e.g., a chairman is above a president, a president is above a vice-president, etc.). These qualities make for easy-to-understand examples.

Three hypothetical corporate charts (representations) are shown in Figure 1-3. The nodes in each F-tree represent officers of the corporation. The links connecting the nodes are the fundamental relations (F-rels) that specify the structure of the hierarchy. These representations are fed directly to CORPORATE-RESEARCHER, one at a time, so that the program can incrementally incorporate them into its knowledge base.

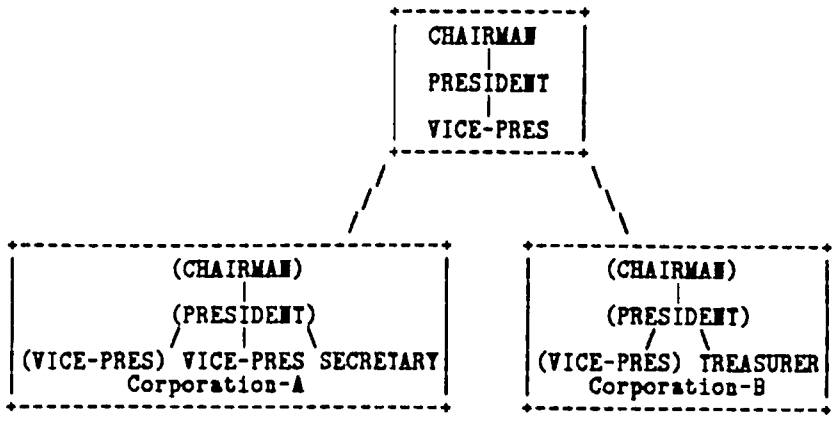


The corporate hierarchies for three hypothetical corporations are shown. All three F-trees show the fundamental relation that binds the members of the hierarchy together.

Figure 1-3: Three corporate charts.

Initially, Corporation-A's F-tree is all that is available to the program. Since a MERGE-based system learns only from information that it has gathered from multiple instance F-trees, there are no possible generalizations that can be made given a single instance. Thus, the initial knowledge base is just Corporation-A's F-tree.

Upon receiving as input Corporation-B's F-tree, CORPORATE-RESEARCHER begins the process of generalizing, modifying representations, and incorporating the new instance into its knowledge structures. Corporation-A and Corporation-B have a chairman, president, and vice-president (VICE-PRES) in common. Therefore, generalizing them together will create a concept of a corporation that has these members in the same ordered hierarchy. The original instances, Corporation-A and Corporation-B, become *variants* of this generalized concept. Figure 1-4 shows this. Each box contains an F-tree. The top box is a generalized F-tree, while the bottom boxes are instance F-trees. The boxes themselves are nodes in the G-tree that describes corporations.



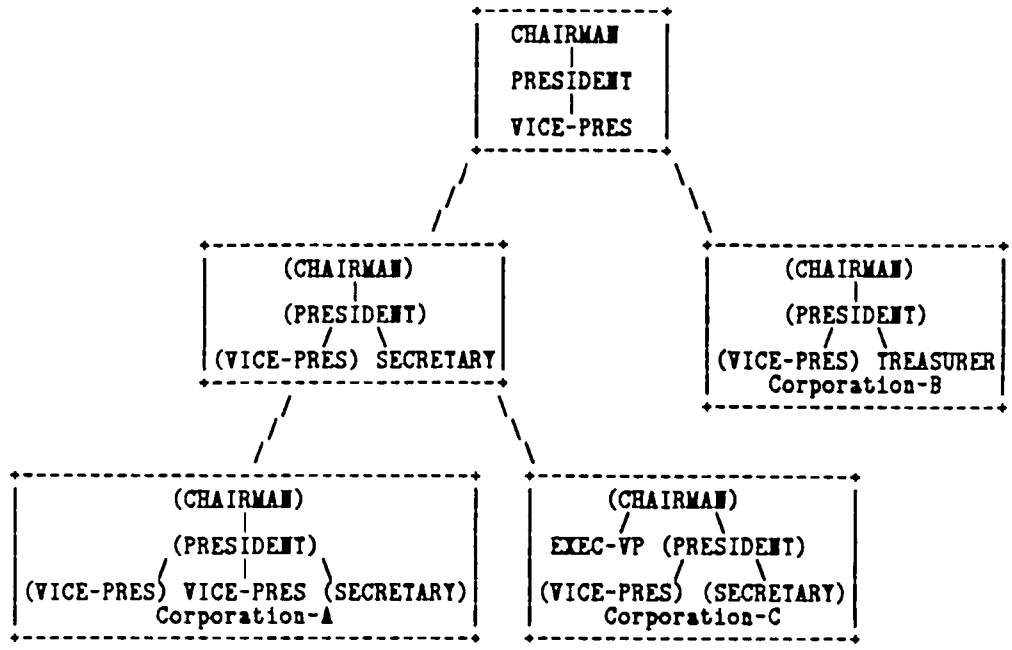
The generalized concept of a corporation (the top box) contains the common elements of Corporation-A and Corporation-B. The original instances (lower boxes) are indexed as variants of the generalized concept. Corporate members that have been inherited from the generalized concept of a corporation are shown in parentheses.

Figure 1-4: The two corporation knowledge base.

There is no need to re-represent parts of a hierarchy that are in common between a variant node and its parent. In this example, the chairman, president, and vice-president have been *inherited* from the generalized concept of a corporation. (The inherited data is shown in parentheses.) Inheritance of this information is used to eliminate the repetition of common elements -- only differences need be stored. MERGE makes heavy use of this operation and augments it with other operations that allow for modification of inherited parts. They are too detailed to go into here, but will be elaborated upon in Chapters 3 and 5.

We can see the use of a MERGE-based system as an automatic means for classifying hierarchies when the representation of Corporation-C is incorporated into the knowledge base. The structure of Corporation-C most closely matches that of Corporation-A in that they both have a secretary and a vice-president below the level of the president. CORPORATE-RESEARCHER recognizes this (i.e., it finds the closest matching instance or generalized concept to Corporation-C according to a metric for matching trees that appears in Appendix A) and builds another generalized concept (F-tree) of a corporation that is more specific than the first one it created. It then indexes both Corporation-A and Corporation-C as variants of this new concept. The new concept is in turn indexed as a variant of the first generalized concept. This creates a G-tree that serves as a classification hierarchy for the instances fed into the system. The resulting G-tree is shown in Figure 1-5.

Concurrent with the process of creating a generalization hierarchy that classifies entire corporations, CORPORATE-RESEARCHER also builds generalization hierarchies that classify each sub-hierarchy that comprises a corporate structure. This process is not an additional task, but rather a consequence of the way MERGE works. Figure 1-6 shows the president G-tree that was created for the



All three instance corporations are classified in this generalization hierarchy. A new generalized concept has been created (middle left box) which is a representation of a corporation that has the common elements of Corporation-A and Corporation-C.

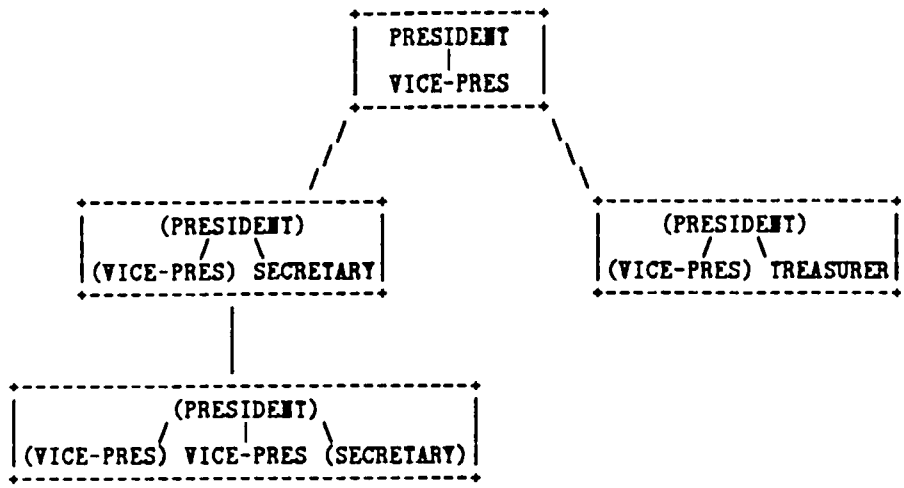
Figure 1-5: The three corporation knowledge base.

same three instance corporations. It has one less node than the generalization hierarchy shown in Figure 1-5 because there is no difference between the president in Corporation-C and its generalized concept. However, the chairman of Corporation-C has an executive vice-president (EXEC-VP) that the generalized concept does not have. In general, G-trees will have different structures depending on the information that they classify.

In Figure 1-6 we can see that the concept of a president that has a vice-president and a secretary reporting to him has been created. This concept may be useful for disambiguating future instance F-trees that are unclear as to who reports to the president. The same can be said for the corporation as a whole. The concept that a corporation has a chairman, president, and vice-president may help in determining the correct ordering of corporate officers for some instance F-tree that does not use the same names for executives that were used in these examples.

1.4 Originality of our work

This thesis is both a synthesis of several recent ideas in knowledge representation and a step forward in hierarchy understanding. We posit a unification of formal representation methods with generalization techniques in a generalization-based memory scheme designed to understand hierarchically structured objects. Assertions



This G-tree was created during the same process of incorporating instances into memory as was the G-tree shown in Figure 1-5. It classifies presidents in the same way the entire corporation is classified. A G-tree is created for each unique member of the instance hierarchies that have children below them in their F-tree (i.e., non-leaf nodes).

Figure 1-6: The president generalization hierarchy.

about the originality of this work fall into three areas: knowledge representation, generalization, and hierarchy theory (i.e., the study of the nature and behavior of hierarchical systems).

The assertions to be made about knowledge representation and generalization are the most significant in this thesis. The MERGE scheme and its implementation in CORPORATE-RESEARCHER and RESEARCHER support these assertions. Our assertions about the nature of hierarchies are more speculative. In the following two sections we will enumerate these points in their order of importance as they relate to this thesis.

1.4.1 Knowledge representation and generalization

1. When representations of individual instance hierarchies are properly integrated with generalizations of these instances, a mutual enhancement of both representation and generalization can occur. The proper integration is obtained by using a type of GBM scheme that we call MERGE. In MERGE, representations of instance hierarchies are encoded in terms of previously generalized instances allowing for the inference of missing data, and disambiguation of contradictory information. These instances, in turn, improve the quality of the generalizations. This feedback between representation and generalization is the essence of the MERGE scheme.

   We will show exactly how this can be achieved by structuring memory

as multiple hierarchies of generalizations (G-trees) based upon representations of instance hierarchies. It will be argued that this approach to organizing memory is cognitively accurate and results in an improved method for understanding, compared to conventional methods in which representation and generalization are treated as separate processes.

Classically, a researcher might set out to build an understanding system by first choosing a representation formalism to capture the data at hand. While working on generalization problems he may then find that his initial choice of representation formalism is not completely satisfactory for making generalizations. The next step would be to refine the representation scheme and try the generalization process again. This test-refine-test strategy is an example of *methodological feedback*.

Methodological feedback is useful for determining a good formalism to use. However, once a formalism is decided upon the processes of representation and generalization must function on their own. The results of a good generalization do not help in the representation of a new instance.

The feedback cycle in MERGE is different. A representational formalism is predetermined for a specific domain (actually, it is basically the same for all hierarchical domains, only relations are domain dependent). However, memory is structured such that instances and generalizations are represented in terms of previously created generalizations. The feedback cycle consists of generalizations affecting the representations of objects and representations used to create new generalizations. This is an example of *internal feedback* -- not methodological feedback. A MERGE-based system dynamically changes its overall representation of knowledge without human intervention. However, the representation language is kept constant.

2. The MERGE scheme offers an effective way to create generalizations about hierarchically structured objects. Researchers have used hierarchies of generalizations as knowledge structures for many tasks. However, the data that their systems generalize about has generally been *non-structured* (i.e., it is a data set -- not a complex structured hierarchy).

We will present a scheme for creating generalization hierarchies based upon *structured* data. Specifically, the data is hierarchically structured -- which we believe to be one of the most common forms of structured data found in both natural and artificial systems. Thus, our scheme can be used in AI systems that heretofore have been limited in the kinds of information they can process.

3. Our approach to understanding objects is well-adapted to real-world situations. Information usually presents itself over a period of time, not all at once. Hence, an intelligent system must be able to learn incrementally by continually reorganizing its memory. A MERGE-based system is designed to make generalizations incrementally as new instance hierarchies are brought into it. Because it is continually creating new generalizations and modifying old ones, its memory scheme must be dynamic. It can not (and need not) be known a priori how the resultant generalization hierarchies will appear. The structure of MERGE's knowledge base is solely dependent on the data it is fed and the order in which it is presented.

   Several other recent systems also make use of a dynamic memory that works by creating generalizations incrementally. Of note are IPP [Lebowitz 80], CYRUS [Kolodner 80], and UNIMEM [Lebowitz 83c]. All of these programs make generalizations of the data they receive as input. MERGE-based systems are unique in that they dynamically reorganize memory while incrementally learning about complex, hierarchically structured objects. These other systems deal with less structured (in the case of CYRUS) or non-structured information.

4. An automatic hierarchical classification of complex objects is achieved by using the MERGE scheme. Each instance hierarchy that is input to a MERGE-based system is encoded according to the fundamental relation links joining its elements together, along with other information. These representations of complex objects are then incorporated into a knowledge base in which they become the leaves of generalization hierarchies (G-trees). A generalization hierarchy exists for each unique object sub-hierarchy in the instances. Thus, MERGE creates multiple classification hierarchies (G-trees) depending on what objects are in the context of the domain under study. This categorization is based on the fundamental relation and other information that augments the descriptions of the instance objects.

   Although classification hierarchies are commonly used in many areas of AI, they are almost always created by humans. The remaining ones do not classify complex, hierarchically structured objects. Our scheme will automatically (without human intervention) build classification hierarchies of arbitrarily complex objects. Furthermore, these classifications are not based on numerical data (as some automatic taxonomy systems use) but rather on the structure of the objects (along with supplemental data), which we believe is more cognitively accurate. The classification hierarchies created can also be used in conjunction with other understanding tasks (e.g., language understanding or problem solving).

1.4.2 Hierarchy theory

1. Using hierarchies of generalizations is a powerful method for understanding hierarchically structured objects. The major advantage to using a hierarchy of generalizations as opposed to some other means of comparing instance objects is that knowledge is grouped into small "chunks" [Miller 56; Rosenbloom and Newell 83]. Because each chunk of knowledge is organized under a node in a hierarchy, this node represents a generalized concept of the information it classifies. Comparisons of new objects can be made against this generalized concept instead of against all of the instances under it. This effectively minimizes the time needed to learn new information.

   Chunking of knowledge is a relatively old idea. However, the use of multiple generalization hierarchies to chunk information encoded in hierarchically structured objects is new.

1.5 Thesis preview

This section provides the reader with a map of the rest of this thesis.

Chapter 2 gives a synopsis of research related to our own. We show where this work fits into the disciplines of AI, systems engineering, and cognitive psychology. The three subject areas that the MERGE scheme bears on are knowledge representation, generalization (learning), and hierarchy theory.

Chapter 3 serves as a technical introduction to the thesis. The basic concepts behind MERGE's representational formalism and generalization techniques are described. Integrating representation with generalization is introduced by showing how inheritance and other operations are used to achieve a GBM. A compact notational scheme is presented that will be used in later chapters to exemplify the details of MERGE and in discussing various issues in representation and generalization.

Before giving a complete description of MERGE and demonstrating how it works, several issues in both representation and generalization are explored. Issues having to do with representing individual hierarchies are investigated in Chapter 4, including more about fundamental relations, non-fundamental relations, and other data RESEARCHER's non-fundamental relation representation scheme is demonstrated as an example of a sophisticated means of augmenting a hierarchy's representation.

Generalization issues are described in Chapter 5. Different types of generalizations are discussed, as is the usefulness of and problems with inheritance. Much of the chapter is spent considering aspects of when to create generalizations and how to use them. All of this is, of course, in the context of hierarchical object understanding.

The contents of Chapters 3, 4, and 5 come together in Chapter 6. The qualities of an ideal MERGE-based system are described. Both the CORPORATE-RESEARCHER and RESEARCHER programs are demonstrated with sample runs. The chapter concludes with a critique of how well these implementations perform relative to an ideal system.

In Chapter 7 we summarize the main points of the thesis and discuss directions for future research.

Although this research is properly classified in the area of knowledge representation within AI, it is related to work in other disciplines and subject areas. The fields of cognitive psychology and systems engineering, as well as AI, have made contributions to hierarchy understanding. The areas of knowledge representation, generalization (learning), and hierarchy theory all bear on our work. Semantic nets and frames are the two fundamental knowledge representation formalisms that are appropriate to hierarchy understanding. We have developed a high-level frame-based scheme similar to MOPs. Research in generalization can be grouped into three overlapping classes: numerical, inductive, and conceptual. The MERGE form of generalization falls mostly into this latter class. As such it allows for generalizations to be made that appear to be "cognitively accurate" in human terms. The major contribution from hierarchy theory to this thesis is the idea of near-decomposability. This concept states that the components of a hierarchical system interact less strongly than the members within any one component.

2. Related Work

2.1 Introduction

Our research is related to previous work spanning three disciplines and three subject areas. As mentioned previously, we are seeking to integrate knowledge representation and generalization into a unified approach to understanding hierarchically structured objects. The three subject areas that are relevant to us are knowledge representation, generalization (learning), and hierarchy theory. Three disciplines that have made significant contributions in these subject areas are artificial intelligence, systems engineering, and cognitive psychology.

This chapter is divided into three major sections. In these sections we present a brief account of related work from each of the three subject areas. Within each of these areas the contributions from the appropriate disciplines are discussed. Before proceeding, we give an overview of where this thesis fits in among the related work that will be surveyed.

The issue of how to best represent knowledge has always been of paramount importance in AI, and consequently there is a large body of research that one can refer to. Learning has also been widely studied by many AI workers. The type of learning that we address, generalization, has been given much attention recently, and so there exist some relatively new papers and programs related to our work. Although several AI systems make use of hierarchies to represent knowledge, none of them have focused on the question of understanding hierarchies per se. Since this is exactly what we are investigating, there is little research to cite that is similar to ours, but there is some related work.

Figure 2-1 summarizes the quantity of available research in each of the nine categories shown. Our work spans all three subject areas within artificial intelligence. Therefore, in the following mini-surveys, special consideration is given to this endeavor. Somewhat less coverage is allotted to systems engineering, and cursory attention is paid to cognitive psychology.

	Discipline		
Subject Area	Artificial Intelligence	Systems Engineering	Psychology
Knowledge Representation	much	n/a	some
Generalization (Learning)	some	some	much
Hierarchy Theory	little	little	little

This table gives a qualitative indication of where research relevant to that presented in this thesis has been done. Our work encompasses all three areas within AI, but we consider it to be primarily in the area of knowledge representation.

Figure 2-1: Related research summary.

Although our work is in AI, we recognize that other fields within cognitive science (much of AI being part of cognitive science) often contribute useful concepts that an AI program can embody. The fields of hierarchy theory and cognitive psychology are of particular relevance to this thesis. Ideas developed in the former field can assist in the overall structuring of a hierarchy understanding system. While the later field can provide insights into how humans understand hierarchies, which can be incorporated into a program, hopefully improving its performance.

Hierarchy theory is properly considered a branch of systems engineering. It is the study of the underlying principles of hierarchies. The goals of this discipline are to develop theories that can explain and predict how hierarchical systems behave -- not simply to enumerate or use hierarchies for representing systems. Even though it has been around for a generation or more, most of the research to date consists of speculative papers about the nature of hierarchies. Nevertheless, there are some important and useful concepts that have been posited. Systems engineering has also harbored the technique of numerical taxonomy or clustering, a way of automatically categorizing data sets, which is a form of generalization.

Human learning is one domain of study of cognitive psychologists. Researchers in AI often point to theories in cognitive psychology for justification of their methods, and several key ideas in knowledge representation have come from psychologists (Miller's "chunking" theory [Miller 56], for example). However, this school's contributions are somewhat peripheral to the issues addressed here and therefore will mostly be referenced in support of other concepts described.

2.2 Knowledge representation

Knowledge representation comprises a wide range of ideas, theories and methods for encoding information about objects and events. The focus of this investigation will be on those schemes that have importance to representing hierarchically structured objects. Our approach will be to first examine formal representation systems, then specific systems that are geared toward representing objects in a hierarchical form. The MERGE scheme is both a formal representation system and one that deals with hierarchical object representations.

2.2.1 Formal representation schemes and systems

Two major formal representation schemes that lend themselves to representations of structured objects are semantic networks [Quillian 68] and frames [Minsky 75]. Variations of these general schemes abound and the distinction between them is often blurred. Nevertheless, one can usually identify the roots of any of these schemes and classify a given representation system as frame-based or semantic network-based. Although our scheme is frame-based, there is much to be learned from a study of semantic networks, as well as frames.

Semantic networks (or nets) were the first of these two representation formalisms to be used in computer programs. An early, integrated natural language processing (NLP) program, SHRDLU [Winograd 72], used semantic nets to encode declarative knowledge about a blocks-world. (We will come back to this program later, as it was a landmark program for representation techniques.) Semantic nets are arbitrarily complex networks in which nodes represent actions, ideas or, in the case of SHRDLU, physical objects. Arcs connecting nodes represent relations among them. For example, if there is a pyramid on top of a block, where the pyramid is represented by a single node and so is the block, then an arc connecting them would represent the relation SUPPORTED-BY.

IS-A links (arcs) are used to represent the concept that one node is an instance of another. For example, *a dog* IS-A *mammal*. All of the properties that a mammal has can be *inherited* by the concept dog (unless overridden). Thus, if the network had the fact that a mammal breaths air encoded in it, then it would be assumed that a dog also breaths air. The word *type* refers to a concept used in a semantic net (e.g., mammal) while the term *token* is identified with an instance of a type

(e.g., dog, if mammal is the type). Inheritance, modified by other operations, is a crucial part of the MERGE scheme. Early semantic net-based systems made only simple use of inheritance (i.e., a token would inherit all of the information in its type). Recent schemes like KL/ONE [Brachman 79a] (described later in this section) and MERGE do a great deal more with inheritance, as will be seen in Chapters 3 and 5.

Any chosen relation can be represented by arcs in semantic nets. Aside from static physical relations, like SUPPORTED-BY, and classification relations, like IS-A, more complex relations, like MUST-BE-SUPPORTED-BY and CAN-NOT-BE-A, are possible. Thus, *a mammal* CAN-NOT-BE-A *reptile*. The deductive reasoning procedures in SHRDLU make use of these relations. Such relations are similar to the ones used to augment a hierarchy's meaning in MERGE.

Much has been written about semantic nets (see [Woods 75; Barr and Feigenbaum 81], for example). They have been and perhaps still are the dominant knowledge representation system used in AI. SHRDLU exemplified the best points about semantic networks, in addition to being an NLP demonstration program. The simple node-arc formalism provides for easy representation of associations. They are useful for encoding static factual knowledge and are versatile in that they permit a wide range of data to be encrypted. Because of the limited domain of knowledge needed to understand the blocks-world, few of the difficulties and limitations of this scheme surfaced [Wilks 74], which is one of the reasons why SHRDLU was so successful. Among the shortcomings of classical semantic nets are: no universally accepted semantic definitions for links; difficulty in representing time dependent knowledge, little distinction of more important data (links) from lower-level knowledge (i.e., all links have equal priority).

One way to overcome the inability of most semantic net representation systems to deal effectively with large networks of data, is to ***chunk*** information into regions within the network and treat these chunks as if they were individual nodes. Thus, a large semantic net with 10,000 nodes could logically be reduced to a network of, say, 200 chunks in which each of the 200 chunks would contain sub-networks of a small size. This ***partitioning*** of a semantic network was proposed in [Hendrix 79].

The idea that humans chunk knowledge was first introduced in [Miller 56]. He suggested that knowledge is organized as a hierarchy of chunks, each chunk serving to index several other chunks, until some base-level is reached. More recent work goes further than this, contending that most human learning not just knowledge representation occurs via this chunking process [Rosenbloom and Newell 83]. We believe a representation scheme intended for use in computer systems to be more "cognitively accurate" if it organizes its knowledge in terms of chunks. Partitioned semantic nets are an example of such a scheme, as are frames.

Several advantages over simple semantic nets are apparent in Hendrix's scheme.

By separating low-level knowledge from high-level knowledge, the encoding process can represent more varied information. For example, the color, shape, and size of an object could be linked together within a partition and the partition itself could have links to other nodes or partitions (e.g., indicating higher level facts about the object's purpose). This hierarchical partitioning results in smaller numbers of objects at any one level that need to be manipulated. In addition, such a scheme comes closer to being cognitively accurate, in human terms.

Frames are another way of solving many of the same problems as partitioned semantic nets. Frames are conceptual objects that are used to group pieces of knowledge into logically consistent blocks. They are most easily thought of as an extension of semantic networks where each node is a comparatively large structure that contains enough information to adequately describe an item at some level of detail. While a node in a semantic net usually is simply the name of an item, a frame can possess information about how to classify an item, how to use it, what attributes it has, and virtually anything else that might be useful to know about an event or object. Furthermore, the knowledge encoded in a frame need not be static (declarative). It may be dynamic (procedural), or it can be a combination of these [Winograd 75]. In either case, a frame should be viewed as "a specialist in a small domain" [Kuipers 75].

If an airline reservation system (see [Bobrow et al. 77] for a description of such a system) used a frame to represent each date on which a plane reservation was made, it might have *slots*[1] in the frame as follows:

```
YEAR:
MONTH:
DAY-OF-MONTH:
DAY-OF-WEEK:
```

The YEAR, MONTH and DAY-OF-MONTH slots might be filled with static data (probably single numbers). The DAY-OF-WEEK slot might contain procedural knowledge as follows:

```
(If  YEAR and MONTH and DAY-OF-MONTH are filled
       then (FIGURE-WEEKDAY))
```

Semantic nets, and particularly partitioned semantic nets, offer a possible formalism for a hierarchy understanding system. However, we have chosen the frame formalism for use in MERGE. In our view, it provides a cleaner and more easily understood approach for building large scale memory organizational systems

---

[1]The term *slots* refers to the "important elements" in a frame [Winograd 75]. Slot fillers can be thought of as references to other frames, which is what Minsky originally proposed. In any particular application of a frame system, a considerable amount of thought must be given to how many slots should be used and what they should contain.

than does partitioned semantic nets. The reasons for using frames are explored more fully in Chapter 4.

One very important aspect of the use of frames as a knowledge representation scheme is the default filling of slot values of token frames from type frames. Default values for frame slots can be easily set up by placing them in a type (or *stereotype*) frame and programming a system so that if no value for a particular slot is specified, then it is inferred from the generic frame. For example, if the YEAR was not explicitly given in the date frame (shown above) then it would be reasonable to assume that the value of the slot should be the current year (as most airline reservations are not booked too far in advance). (In this case, the stereotype date frame would have to have its YEAR slot filled.) However, if the DAY-OF-MONTH was not given, it would obviously be a mistake to assume some value from a stereotype (assuming that only a few reservations are made on any given day and that there is no good reason for choosing a particular default date).

In order to effectively use frames as a representation system, several other operations, aside from default processing, are essential. These include: matching one frame against another, allowing for inheritance of properties from higher level frames, type checking the values that can fill a slot in order to ensure that only valid ones are accepted, and general abilities to manipulate a connected network of frames. KRL [Bobrow and Winograd 77a], a language that was developed specifically to allow for knowledge representation in the form of frames, includes facilities for these functions among others. Many of these functions, particularly matching and inheritance, are of importance for use in systems that perform some sort of generalization about their knowledge.

KL/ONE [Brachman 79a] and FRL [Roberts and Goldstein 77] are two systems that are similar in purpose to KRL. But they go beyond it by imposing certain structuring rules that make it easier for researchers to develop systems. In particular, both KL/ONE and FRL embody the idea of inheritance hierarchies as their very nature. Inheritance hierarchies are equivalent to our G-trees. Object hierarchies and generalization hierarchies can be represented in these systems. Although we do not use either of these formalisms (because of our need to integrate representation and generalization via feedback) they are worth taking note of here.

KL/ONE is both a language (embedded in LISP) and a methodology for organizing partitioned semantic networks. Objects represented in KL/ONE are structured much like they are in a frame-based scheme. However, KL/ONE's structural formalism also provides a way of establishing inheritance hierarchies. A distinction is made between stereotypical objects and instantiated ones. Thus, the properties of an object can be attached either to a stereotype for that object or to the object itself. Because of the hierarchical nature of KL/ONE, complex but well-organized inheritance dependencies can be established. By using a limited set of possible links, the semantics of the network are clearly defined.

FRL is much like KL/ONE, but instead of imposing restrictions on the semantics of links, it forces the network of frames to be hierarchically connected. All frames must be joined together using INSTANCE and A-KIND-OF links. Therefore, the representation tree (actually a tree-like network) has as its root the most general frame and its leaves are the lowest level instances of whatever the network is representing. For example, if one were representing car models, the root frame might be all automobiles; below that, frames encoding General Motors, Ford, and Toyota cars; at the bottom of the tree there would be Skylarks, Mustangs, Celicas, and so forth. The A-KIND-OF links point backward, so that Buicks are A-KIND-OF General Motors car. Unless otherwise specified, Buicks would inherit all the properties that are in the General Motors frame. This type of representation is very helpful in forming and storing generalizations made about objects or events.

Frames or partitioned semantic nets linked together into hierarchical structures are representational formalisms that lend themselves to generalization processing. INSTANCE and IS-A (although we will call them VARIANT and VARIANT-OF when used in G-trees) links correspond to specialization and generalization, respectively. Many representation/generalization schemes use this basic formalism in constructing complex network descriptions of physical objects. Our work departs from these schemes in that representation doesn't "lend itself to generalization" -- the representation that MERGE uses integrates generalization into the way objects are encoded. In MERGE, after an object's representation is incorporated into memory the resulting generalization is used to modify other existing and future representations.

The NETL scheme [Fahlman 79] deserves mention here. NETL is a formal representation system that has a wide range of applications. Of particular interest to us is its recognition of the importance of the interaction between representation and generalization. Unfortunately, Fahlman's work does not describe how to integrate these two processes, but he does present an example that is similar to what we are working on using PART-OF and IS-A hierarchies to represent objects.

2.2.2 Hierarchical systems

In this section, we look at some specific systems that either make use of hierarchies to represent data or are suitable for use in a system that does.

KL/ONE, FRL, and KRL have been used to implement various systems that use hierarchical knowledge structures. Among them are GUS [Bobrow et al. 77], a program designed to provide information on airline flight schedules which served as a testbed for the development of KRL. Lehnert's COIL [Bobrow and Winograd 77b; Lehnert 77] was another program written in KRL. It concerned itself with drawing inferences about physical objects. Physical object representation is particularly relevant to this thesis because RESEARCHER understands physical objects by using the MERGE scheme.

Another program, OPUS [Lehnert and Burstein 79] also deals with physical object representation. It employs a scheme called Object Primitives [Lehnert 78] to encode the functional aspects of an object. Although it is not in itself a hierarchical system, it was designed to be an extension of Conceptual Dependency (CD) theory [Schank 72]. CD is primarily a way to represent actions. However, the methodology behind the theory is common to many advanced representation systems. (RESEARCHER's scheme for representing physical relations among objects is functionally similar to CD, and is described at the end of Chapter 4.) CD has been used as the basis for truly hierarchical representation systems, in the form of MOPs, as we will discuss.

CD works on the theory that actions (verbs) can be reduced in meaning to canonical combinations of a small group of primitive ACTs. For each ACT, there are a fixed number of arguments that accompany it. That is, an actor, recipient, object, and other possible *case* slots must be filled for each ACT. For example, "John gave Mary a gift" would have the representation:

```
(ATRANS
  ACTOR: John
  OBJECT: gift
  FROM: John
  TO: Mary)
```

ATRANS, one of the primitive ACTs, is used to represent the meaning of the verb "gave" and indicates Abstract TRANSfer (of possession) of an object.

CD is capable of representing a wide range of actions and situations. In addition to the basic ACTs, both mental and physical states of a being or an object can be encoded. The fact that an event may enable, disable, or cause a state, is also representable within CD. Using these connectives, it is possible to represent the meaning of a series of sentences that comprise a story with one complex CD structure.

The major contribution of CD that is relevant to this thesis is the way it integrates two very useful concepts: *case grammars* [Fillmore 68] and *semantic primitives*. Case grammars were an outgrowth of both classical linguistics and Chomsky's transformational grammar [Chomsky 65]. They reflect classical linguistics in the sense that they identify the various parts of a sentence such as the main verb phrase and noun phrases. However, it is not the surface structure of the sentence that is captured, but rather the verb's meaning. Thus, regardless of the formal structure of the sentence the "case frame" extracted by using case grammars will be the same for sentences employing the same main verb. Structurally, the case frame looks very much like what was presented in the CD example (above) with actor (or agent), object, instrument and a few other slots available. Case grammars classify verbs by the slots (cases) that must accompany a particular verb. For example, if the verbs *open* and *throw* require the same slots (OBJECT, AGENT, and INSTRUMENT) for their case frames then they would be grouped

together. CD goes beyond case frames by defining a system of primitives and rules to manipulate them that captures the meaning of a sentence, rather than having a case frame for every verb.

The second building block of CD comes from both linguistic and psychological research. Semantic primitives are generally defined to be the lowest level of symbolism in a representation system. In practice, an understanding/representation system uses semantic primitives as a way of classifying a set of concepts, such as actions or physical objects. For example, RESEARCHER's relation representation scheme [Wasserman and Lebowitz 83] uses a set of semantic primitives that are designed to decompose physical relations. Five primitives, used in combination, attempt to achieve for physical object relations what CD attempts to do for actions.

CD demonstrated the effectiveness of using a primitive-based representation scheme in conjunction with frames. Many programs have been written that employ CD. These include: MARGIE [Schank 75], the earliest CD-based program; SAM [Cullingford 78], which demonstrated the use of *scripts* in story understanding; and PAM [Wilensky 78], which made use of *plans* and *goals*.

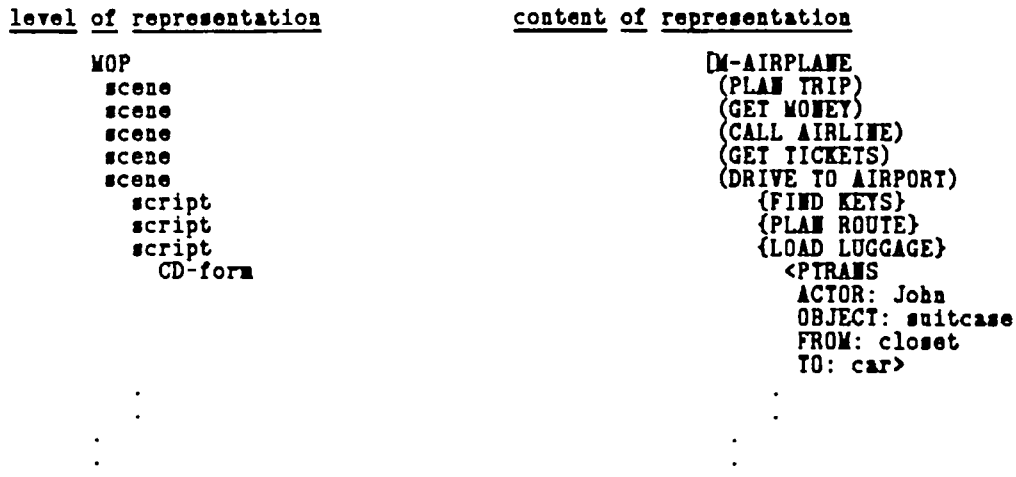
We are interested at looking at higher levels of knowledge representation because that is what MERGE offers. Scripts, plans, goals, and Memory Organizational Packets (MOPs) [Schank 80; Schank 82] are successively more sophisticated representational concepts. MERGE functions at the level of MOPs (it might even be thought of as using MOPs), in that it dynamically reorganizes memory (which is explained below).

Scripts are a way of organizing sequences of events (CD-forms) in memory. They are static structures that are intended to mirror how humans carry out simple activities. Plans offer a higher level of representation. Their purpose is to organize memory such that previously unencountered situations can be understood in terms of known events (i.e., scripts). Thus, a representation hierarchy with CD-forms as the leaves of this hierarchy can be developed. The next step in this representation formalism are MOPs.

MOPs are very high level representational structures that organize scenes, scripts, and supplemental data into a coherent picture of an event. In this sense, MOPs work much like plans, but are more powerful and allow for dynamic script building. That is, MOPs, scenes, and scripts can be collected into a memory/processing structure that fits a particular situation. They can be used in both predictive and understanding modes. When used in a predictive mode, expectations can be made about future events. If an expectation fails the memory structure can be reorganized to account for it by modifying, deleting, or adding a scene, script, or MOP.

To get a better idea of what MOPs can represent, consider the MOP skeleton

shown in Figure 2-2. Here we see that the M-AIRPLANE MOP is composed of several scenes, which in turn contain scripts, which are complex CD descriptions of a simple activity. (These scripts differ from those that SAM uses in that they are instantiations of a scene -- they are not necessarily a rote memory structure.) That is, scenes are at a higher level of representation than are scripts, and MOPs are at a still higher level. This diagram shows only what the DRIVE-TO-AIRPORT scene expands to. All of the other scenes have a script representation as well. Although MOPs are a form of frame, they are far removed from something as simple as the date frame shown earlier.



This figure (adapted from [Schank 82]) shows a representation hierarchy running from CD-forms to MOPs.

Figure 2-2: MOP skeleton.

What has not been shown in Figure 2-2 is the dynamic nature of MOPs. MOPs not only serve to organize static episodes in memory but also to allow memory to be dynamically reorganized to better reflect the state of the world. For example, the M-AIRPLANE MOP can be modified to account for the possibility of shuttle flights. This might take place by reorganizing the scenes so that CALL AIRLINE and GET TICKETS scenes do not have to take place.

There are many programs that make use of hierarchies, in some form or another, to represent objects or events. Usually the knowledge structures embedded in these programs are built by a human expert (in whatever domain is being studied) and used by the system to either solve a problem or store additional information according to previously established classification categories. They illustrate the need for programs capable of creating hierarchical structures without human intervention. MERGE-based systems, like CORPORATE-RESEARCHER and RESEARCHER, offer this capability.

NOAH [Sacerdoti 75] stored its knowledge about assembling physical objects in a hierarchy of plans. SCHOLAR [Carbonell 70] used a semantic net-based hierarchy

of geographic information on South America. TEXT [McKeown 82] served as a natural language query system to a database about ships. It used two hierarchies: one to store generalized knowledge about physical objects (e.g., ships, submarines, etc.), and another to encode attributes of ships according to topics (e.g., speed-indices). A program in [Hayes 77] employed a categorization hierarchy that classified animal body-part hierarchies. It used a generalization (IS-A) hierarchy and PART-OF hierarchies that were built by a human expert in order to implement the knowledge structures it needed.

2.2.3 Summary

Two fundamental approaches to knowledge representation, semantic nets and frames, have been described. Frames are a more recent development and for our purposes appear more capable of representing a large amount of information in a hierarchical structure. We have chosen to use frames in MERGE; however, partitioned semantic nets offer equivalent power.

CD theory is a widely used example of a scheme based on semantic primitives in conjunction with a framed-based memory encoding. This particular combination forms the foundation upon which more elaborate schemes can be built. It also demonstrates how a useful semantic primitive scheme can be developed. MOPs, an advanced form of knowledge representation, are constructed from a hierarchy of sub-structures (scenes, scripts, CD-forms) and are readily reorganized so that MOP-based memory is dynamic in nature.

2.3 Generalization

Generalization is usually thought of as a particular type of learning. It is the process of recognizing commonalities within a set of input examples and building a new concept (or concepts) containing this information. Several possible approaches to carrying out this task are possible. Numerical taxonomy, inductive inference, version space, and conceptual clustering are the names of some of the more formal methods. Many other schemes are used by particular AI systems. In addition, cognitive psychologists have suggested methods that humans use to make generalizations.

This section discusses some of these generalization methods that have application to hierarchy comprehension. We begin with an overview of a few abstract generalization systems. Next, some AI programs that have been successful in their use of generalization are discussed. Finally, a few comments about some particularly important and relevant work in psychology are made.

2.3.1 Abstract systems

Abstract generalization systems are those that are not particularly intended for use in any specific application. They can be divided into three major, non-mutually exclusive classes: numerical, inductive, and conceptual. Numerical taxonomy (clustering, generalization, etc.) involves structuring instance examples into a hierarchy based upon some numerical measure of similarity, sometimes called a distance measure [Ben-Bassat and Zaidenberg 84]. Schemes within the inductive class use rules of inductive inference in a formal way to build conjunctive and/or disjunctive hierarchies of sets. The conceptual class of systems includes any method of generalization where the primary motivation is to develop a categorization system that in some way mirrors human cognitive processes. Our work lies within this class. We use generalization as a means to learn about hierarchically structured objects according to their structure.

In numerical clustering systems (see [Michalski and Stepp 83a], for example), some *a priori* value is assigned to each property that an instance example has. These values are used by a comparison algorithm to determine which examples are closest together (i.e., evaluate to be nearly numerically equivalent). Near neighbors become variants of a common ancestor node in a classification hierarchy. Higher nodes in the hierarchy group either instances or other generalized nodes according to how similar their values are.

Usually this process is accomplished *en masse*. That is, all instances must be present at once and the generalization algorithm builds the classification hierarchy by performing many comparisons among the input examples. Real-world situations, however, often provide data *incrementally*. That is, instances become available to a learning system over a period of time. Therefore, an incremental learning system must continuously reorganize its knowledge base if it is to provide an ongoing representation of the known data. (See [Lebowitz 83a] for a discussion of the needs of real-world intelligent information systems).

Despite this shortcoming, numerical taxonomy is an effective way of categorizing a large number of objects. The MERGE scheme is designed to be used in systems that also operate in domains with many instances. Thus, a numerical taxonomy can serve as a benchmark by which to measure the performance of new cognitively-based generalization schemes, MERGE in particular.

Inductive inference techniques are a way of organizing data according to a strict set of rules.[1] There has been a significant volume of research into inductive processes and much of the mathematical machinery developed can be used to ease

---

[1]There has been a great deal of work on inductive learning and a survey of it is beyond the scope of this paper. Therefore, the reader is referred to several excellent surveys including: [Angluin and Smith 82; Dietterich and Michalski 81; Michalski 83; Mitchell 82].

the programming burden as well as speed up the processing of instance examples. Thus, inductive inference has become a popular base on which to build generalization systems. However, these techniques suffer from many of the same problems that numerical taxonomy does. In particular, they do not necessarily mirror the results of human cognitive processes.

Conceptual generalization methods are usually intended to model some aspect of a theory of cognition. As such, they coincide better than other methods with the types of generalizations that people make, but tend to be difficult to implement (as compared to inductive inference). In general, conceptual generalization systems work by using some means for comparing instances that are described in terms of some *basic* properties. The basic properties should have some correlation to human perceptive levels. Generalized concepts are then created using combinations of basic properties that have been found to be common to one or more instances.

To compare such a system with one using numerical taxonomy, consider the problem of categorizing animals. A cognitive-type scheme might place whales and dogs far away from each other in the generalization hierarchy because they don't *seem* very similar. That is, their basic properties (as humans perceive them) are sufficiently different, so that they have little in common (e.g., whales are very large, live in water, and are usually not kept as pets; while dogs are small, land-based, and man's best friend). But a numerical approach might decide that they are similar in that both whales and dogs are mammals with tails and large mouths (relative to their body size). Of course weighting factors for these features would have to be high for a numerical-based system to get these results.

2.3.2 Specific systems

Most generalization schemes are actually a combination of these three basic classes. Here we examine a handful of such systems.

Conceptual clustering [Michalski and Stepp 83b] uses numerical taxonomy combined with global optimization based on conceptual quality measures. The result is a system that provides more meaningful generalizations (clusters) than would be arrived at by numerical clustering alone. An example from the CLUSTER/2 program in [Michalski and Stepp 83a] demonstrates how conceptual clustering offers advantages over standard numerical taxonomy in classifying brands of personal computers (i.e., they use numerical taxonomy as a benchmark). The CLUSTER/2 program produced various hierarchies in which the highest level branching criteria was microprocessor type, the lower level branches distinguished other factors such as display type, keyboards, and memory configurations. The comparison NUMTAX program, which tried 18 different numerical methods, developed less meaningful classifications for the same input data. (Memory configuration or keyboards took priority over microprocessor type.)

ARCH [Winston 72] was an early generalization program based on conceptual techniques. Using semantic nets to represent the physical objects in a blocks-world, learning about simple object structures (arches in particular) was carried out. An arch may be represented by a simple semantic net. After presenting the ARCH program with a correct example of an arch, subsequent nets were given to the program along with external input declaring each example to be either correct or a near-miss. From this data, the program updated the semantic net representation for an arch. Specifically, the program compared the training examples it was given and extracted the information common to the correct examples that did not contradict what had been learned from the incorrect examples. The incorrect examples that ARCH was fed were picked to be "near-misses" so that its learning was carefully focused.

The type of learning that we are interested in is different from that which ARCH performed. The generalizations that ARCH made were carefully guided by a set of training instances. It was easy to predict what ARCH would learn in advance. MERGE-based systems also form concepts from the data that is fed to them. However, they need not have "training instances"; they will create generalized concepts from whatever information is input. (Of course, they may not always be the "correct" concepts.) In essence, MERGE, not the human systems builder, decides what concepts should be created.

The objects generalized by ARCH were fairly simple compared to those in later programs, such as IPP [Lebowitz 80; Lebowitz 83d]. IPP uses MOPs as long term memory representations of stories it reads about terrorism. It scans stories from wire services and newspapers and understands them in terms of what information it has gathered from previous stories. The use of MOPs residing in memory for understanding the current input text is one of the important features of this program. IPP recognizes similarities and differences between events stored with MOPs it has in memory and then uses this observational data to build other MOPs. This particular arrangement is called *generalization-based memory* (GBM) [Lebowitz 80; Lebowitz 83c; Lebowitz 83d]. MERGE also uses a form of GBM to store its knowledge. Furthermore, MERGE is similar to IPP in that they both create concepts without human intervention.

To exemplify this type of generalization, consider the following (taken from [Lebowitz 80]):

UPI, 4 April 1980, Northern Ireland

> **"Terrorists believed to be from the Irish Republican Army murdered a part-time policeman ....."**

UPI, 7 June 1980, Northern Ireland

> **"The outlawed Irish Republican Army shot dead a part-time soldier in front of his 11-year-old son in a village store Sunday."**

From these stories, IPP made a generalization we can paraphrase as:

"Terrorist killings in Northern Ireland are carried out by members of the Irish Republican Army."

This generalization is made possible by a comparison of MOP slot fillers. The stereotypical MOP for a terrorist killing event has slots for place and actor, among others such as victim, method, etc. The program assumes that all facts it knows about are relevant to compare. After forming this generalization, IPP will use it to make inferences while reading other stories. Thus, if a new story about a terrorist act in Northern Ireland came across the UPI wire, and no mention of who committed the act was made, then IPP would assume that the Irish Republican Army was responsible. This sort of assumption is an example of default processing mentioned in the context of GUS, but carried out dynamically and at a higher level of representation.

Lebowitz's work is not the only recent research into using generalization processes in conjunction with natural language understanding systems. (A survey of several semantic-based natural language processing systems, that make use of generalization, appears in [Wasserman 85].) CYRUS [Kolodner 80], a program developed concurrently with IPP, uses a similar generalization process in order to understand events concerning the activities of individuals (Cyrus Vance was the prototype). They differ in the way that they make use of knowledge gained through generalization and the level of detail they include (CYRUS uses much more). IPP uses its inferred knowledge in order to help itself in understanding further input text, while CYRUS answers user questions by employing this knowledge to help it reconstruct episodes in memory. These reconstructed episodes can be thought of as a re-creation of the "mental state" that the understanding system had while reading the original text.

IPP, CYRUS, and MERGE-based systems (CORPORATE-RESEARCHER and RESEARCHER) have much in common, particularly in the way that memory is organized. They all use some type of GBM approach in which their knowledge base changes dynamically depending on the data that the instance examples provide. MERGE differs from the other two in that it seeks to understand objects, not events. In addition, the objects it understands are highly structured (i.e., are complex hierarchies) while the events the IPP and CYRUS process are less structured.

Other systems that have made use of generalization hierarchies as a means for learning include: ENHANCE [McCoy 82] and Meta-DENDRAL [Buchanan and Mitchell 78].

The ENHANCE program uses generalization as a way to restructure an existing database. It sub-divides entity classes in a database according to a set of world knowledge axioms. These sub-classes form a structured hierarchy that is tailored to

a particular use by the information contained within the axioms. The enhanced database is then used by a text generation program to provide intelligent responses to user queries. The work done by the generation program is simplified because most of the inferencing it needs to perform has already been pre-computed by ENHANCE. However, it is not a dynamic learning system in the sense that IPP and CYRUS are.

Rule learning is the term that Mitchell applies to his notion of ***version space*** [Mitchell 77]. Version space is a representation/generalization method for finding the set of all possible rules that can account for the outcome of some particular action given the results of this action. The representation it builds is a hierarchy of rules extending from the most specific to the most general. They are used in a program called Meta-DENDRAL which learns rules for use in the production system that DENDRAL [Lindsay et al. 80] uses. Meta-DENDRAL uses a dual form of generalization based on the version space method. It can produce production rules that are as general as possible, but still fully account for the observed data, or it can produce very specific rules, or both. Although the MERGE scheme always produces the most specific generalizations that it can, version space suggests interesting ways to expand upon MERGE in applications which may need alternative generalizations.

2.3.3 Cognitive processes

There has been much work done in psychology in human cognitive modeling (see [Kintsch 77] for an overview). As a consequence of this work, many different ways of thinking about generalization have emerged. Some researchers believe that all learning is in some way generalization, while others reserve the term "generalization" for a specific cognitive process, such as building stereotypes from a limited number of examples. This is essentially the definition that we have adopted.

[Rosch et al. 76] have investigated the existence of fundamental classes of ***basic objects***. They present evidence which shows that there exist natural categories of objects that people use while perceiving physical objects in the real world. Basic objects are but one level in a hierarchy of perceptual levels. This level is the one at which humans form cognitive pictures of the real world. For example, a car is considered to be at a basic level in the hierarchy that has the following order: vehicle - car - 4 door sedan.

In subsequent research, [Hemenway and Tversky 84] claim that part configuration of objects (F-trees in our work) is the underlying reason for classifying an object as *basic* or otherwise. They demonstrate that the basic object level is the one that people describe in terms of components. For example, cars tend to be described by their constituent parts (e.g., body, chassis, engine) more readily than a vehicle or a 4 door sedan is. From their studies, they conclude that part information is "more

salient in the minds of people when they think about entities at the basic level." Thus, this work seems to support our use of part decompositions as the fundamental means to understand hierarchies. Of course, this assumes that MERGE will be used to understand objects at a basic level.

The idea that semantic primitives or basic objects lie at the root of human understanding is also brought out in the ***chunking*** theory of learning [Miller 56; Rosenbloom and Newell 83]. Each chunk is composed of other chunks until some base level (i.e., semantic primitive) is reached. Miller pointed out that there is a size limit (seven, plus or minus 2) to the number of sub-chunks any one chunk in human memory can contain. Thus, the issue of a limited ***span*** in human memory hierarchies is raised. MERGE-based systems tend to have small span G-trees (see the examples in Chapter 6) because new generalizations are created each time F-trees are compared and found to have some elements in common. The idea of span in a hierarchy is described in the next section.

We have mentioned the concept of chunking earlier in this chapter (when partitioned semantic nets were described). A hierarchy can be thought of as a manifestation of the chunking process, carried out repeatedly. Each node in a hierarchy acts as a means to collect all the children of that mode into a chunk. Each child in turn acts as a chunking device for its children, and so on. Since MERGE employs both object hierarchies and generalization hierarchies this concept is important to our work. Many of the benefits of chunking information are exploited in MERGE, as is described in later chapters. We believe that the ability of MERGE-based systems to readily chunk data is akin to human cognitive processing of hierarchies.

2.3.4 Summary

Generalization is a kind of learning that allows instance examples to be compared and have their common information extracted. Three classes of generalization schemes exist, with most real systems being combinations of these. Numerical taxonomy is a way of organizing data according to assigned numerical values, usually into a hierarchical categorization. Induction is a formal mathematical means of structuring input examples in conjunctive and/or disjunctive hierarchies. Conceptual generalization strives to capture elements of human cognitive processes. Our scheme is an example of this later class.

Several important properties of generalization systems are: whether they work incrementally or all-at-once; how they integrate with representation schemes (e.g., semantic nets, MOPs, etc.), and whether they produce cognitively plausible generalizations (in human terms) or not. The concepts of semantic primitives and basic objects imply the existence of base levels of understanding from which humans (or machines) construct generalization hierarchies.

2.4 Hierarchy theory

We turn to hierarchy theory in an attempt to identify what a hierarchical knowledge structure is all about. Over the past couple of decades, a handful of researchers have given considerable thought as to why both natural and artificial systems are hierarchically organized.

Although an integration of generalization with representation is a powerful hierarchy understanding method (as we will show), it is useful for a systems designer to have a thorough comprehension of the domain of application. First, a brief look at some fundamental concepts of hierarchy theory, is taken. The concepts that we mention come from AI but most of the researchers cited would probably describe themselves as cognitive scientists. Next, the field of general systems theory is discussed. This discipline has existed within systems engineering for some time and has made a serious attempt at quantifying how complex systems, including hierarchies, function.

2.4.1 Fundamental concepts

A well-known work on hierarchy theory within the disciplines comprising cognitive science is a paper written by Simon called *The Architecture of Complexity* [Simon 81]. In this document and others ([Simon 73], for example) he posits that complex systems are usually hierarchical in nature and that they share certain common properties. The fundamental goal of hierarchy theory is to discover and elaborate upon these properties.

Simon identifies one major property of hierarchies, that of ***near-decomposability***. A nearly-decomposable system is one in which the interaction among the components that make up the system is weaker than the interactions that keep any one component intact. The contention is that systems can evolve in complexity by making use of this property and that a hierarchy is the natural form that a complex system usually develops into.

What we call the ***fundamental relation*** (F-rel) of a hierarchy relates to the idea of near-decomposability. F-rel links join the nodes of a hierarchy together according to the order of subservience of nodes (more about this in Chapters 3 and 4). There are more F-rel links within any one sub-assembly than there are links joining this sub-hierarchy to the entire hierarchy. Thus, each sub-hierarchy is a more tightly bound unit than the whole. Hence, the hierarchy is nearly-decomposable -- it would be completely-decomposable if no links existed joining sub-hierarchies together. Other, non-fundamental relations also contribute to making a hierarchy less decomposable.

To clarify this concept, consider how large corporate structures may develop. Initially, a company might start out with a president and a handful of employees

reporting directly to him. As the business grows, it is found that too many people need to talk with the president so that his productivity falls. Intermediaries (vice-presidents) are interposed between the workers and the president thereby reducing the presidents' *span-of-control* by partitioning the corporate structure. The company can be said to be nearly-decomposable in the sense that interaction within a single department (created by this partitioning) is greater than the interactions among different departments.

The notion of span-of-control, or simply span, is common to all hierarchies. It refers to the branching factor out of a node in the hierarchy. Some hierarchies are *flat* in that they have a broad span (many branches) and little depth, while others are more deep than wide (e.g., family trees). As mentioned previously, the G-trees the MERGE creates have a small span. This is due to the fact that when F-trees are compared they usually have something in common causing a new generalization to be built. The constant building of generalizations is analogous to interposing intermediaries in a corporate structure. The result is a deeper, rather than wider, hierarchy. This seems to correlate well with models of human cognition (e.g., chunking). It is an important consideration and will be discussed further in later chapters.

Most researchers who have written about hierarchy theory acknowledge this concept of near-decomposability. An extension of this property of hierarchies has been identified by several workers [Sussman and Steele 80; Fahlman 79; Shlichta 69]. Variously termed *almost-hierarchies* and *tangled hierarchies*, the idea is that some complex systems are not strict hierarchies (i.e., trees). They are actually a superimposition of several trees in which a node can appear in more than one tree (e.g., directed acyclic graphs). For example, a cylinder of a car's engine might be considered a part in the automobile's structural hierarchy (i.e., part of the engine-block which is part of the motor, etc.), while at the same time being a component in the functional hierarchy describing how a car converts chemical energy into motion. The near-decomposability of both the structural and the functional hierarchies remains intact. That is, the cylinder's parts (e.g., the piston rings and cylinder-head) interact more strongly than the cylinder does with, say, the windshield wiper.

We recognize the existence of tangled hierarchies in complex systems. However, for many applications of MERGE describing a hierarchy with a single F-rel is sufficient. Information that would otherwise be carried by tangled hierarchies can be embodied in the single hierarchy by using non-fundamental relations among arbitrary nodes in the single F-rel representation.

2.4.2 General systems theory

Several ambitious attempts at developing a rigorous theory of hierarchies come from general systems theory, a branch of systems engineering. As its name indicates, this discipline seeks to arrive at a theory that provides a basis for building more specific theories. These specific theories might apply to unrelated fields, however the general theory would embody deep principles common to all systems. Hierarchies, being a frequent form that complex systems take on, could be understood by such a general systems theory.

Several approaches to obtain a general system theory have been tried. [Mesarovic 64] gives a mathematical definition based on set-theoretic principles to which any general systems theory must adhere. Although the details are not immediately relevant to our goals here, the idea that a mathematical basis to a theory of hierarchies exists is certainly intriguing. Perhaps more useful is an axiomatic formulation of a general systems theory [Churchman 64].

Briefly, Churchman's "axioms" are: (1) systems are ***designed*** and ***developed***; (2) systems are composed of components; (3) these components can be systems in and of themselves; (4) systems evolve in a direction of increasing stability (this is also a point that Simon makes); (5) a general system is the ultimate in stability. The remaining axioms are somewhat esoteric and less useful: (6) there exists only one general system; (7) the (general) system is optimal; (8) general systems theory is the search for this single, optimal theory; (9) the search gets increasingly more difficult with time, and it never ends.

The motivation, in MERGE, for maintaining generalization hierarchies, for each unique object in the instance hierarchies, is stated in axioms 2 and 3. In other words, if a hierarchy of generalizations can be used to understand a particular type of object, then generalization hierarchies can also be used to understand the parts of an object.

Churchman's last axiom not withstanding, most researchers feel that there are principles common to all complex systems systems and that they can be discovered and formalized. Unfortunately, no one has yet been successful in coming up with a theory of hierarchies much less a general systems theory. We hope our research will contribute to developing a theory of hierarchies, and therefore help in developing a general systems theory.

2.4.3 Summary

The nature of hierarchies is only beginning to be understood. It is believed that complex systems often take the form of hierarchies and that all such systems share certain common properties. The most clearly identified property is that of near-decomposability. Other concepts, such as span-of-control, are important considerations in describing and understanding hierarchies.

Some researchers state that hierarchies are not necessarily limited to being simple trees; they suggest that tangled hierarchies are the natural way to capture this concept. We recognize that hierarchies are not simple trees, but believe that relations among arbitrary members of a hierarchy is a reasonable way to approximately capture partial decompositions of a system. Thus, we will represent a hierarchical system by a single tree-like structure based on one F-rel with other relations superimposed on it

2.5 Summary

In the above surveys of representation, generalization, and hierarchy theory we have discussed several schemes and methods that have contributed to cognitive science's repertoire of intelligent understanding tools. Most of the work done to date has concentrated on one particular aspect of the task we have before us. Typically, a researcher devises a representation scheme or a generalization technique to solve some particular problem. This is often used as the backbone of an AI program to explore one or more, usually narrow, domains.

A few scholars have gone as far as implementing integrated generalization and representation schemes, but they are not completely suitable for hierarchy understanding for a number of reasons. A hierarchy understanding system that is to be used for real-world knowledge acquisition should have the ability to: automatically build representations without human intervention, incrementally construct generalizations, dynamically reorganize memory to better reflect learned knowledge, make use of the near-decomposability of hierarchies to store information in a compact form, recognize and exploit the inter-relationship of the representation language with the generalization method. Some of the systems in existence have one of more of these characteristics, but none of them meet all the requirements.

We have set the stage for describing the MERGE scheme and related issues in representation and generalization. MERGE borrows ideas from many of the works that have been discussed in this chapter. Most prominent among these are: IPP - for the concept of a GBM, CD - for the basis of RESEARCHER's relation representation scheme, and frames - for the overall knowledge encoding scheme of MERGE. Other research has had a less tangible influence on our work. In particular, the ideas of memory chunking and near-decomposability have been important to us.

A hierarchy is defined by one or more partial orderings of nodes connected by fundamental relation (F-rel) links. The F-rel is a formal characterization of the organizational concept behind a hierarchical system. The representation of a hierarchy by using F-rel links is called an F-tree. MERGE uses a single F-tree augmented by relations as a pragmatic means for representing hierarchical systems. Non-fundamental relations are superimposed on an F-tree allowing a more complete description of a hierarchy to be captured. A generalization tree (G-tree) is used to organize instance F-trees by creating a hierarchy of generalized F-trees (concepts). Several G-trees exist in memory, each one serving to classify a unique sub-tree (object) within the instance F-trees. The entire combination of F-trees and G-trees is called a unified memory structure. It is the heart of the MERGE scheme. A compact notational formalism is introduced in order to easily encode unified memory structures for the purposes of analyzing and demonstrating the details of MERGE.

3. Principles of Hierarchy Understanding

3.1 Introduction

Understanding of hierarchies is a difficult task for an automated system. Such a system must be able to both represent a single instance of a hierarchy and to generalize many representations into a knowledge base. In addition, we hope in some way to model human learning of hierarchical phenomena. Humans do not appear to delineate representation from generalization -- they are part of an integrated process. As such, one of our primary goals is to unify representations of hierarchies with generalizations of the same. In this thesis, we address these matters and present a form of generalization-based memory scheme designed specifically for hierarchy understanding, MERGE.

MERGE (Mutually Enhanced Representation and GEneralization) is a scheme for organizing memory to be used as the knowledge base in an intelligent information system. As its name indicates, MERGE unifies representations of single instances of hierarchies with generalizations of them. There is a mutually beneficial interaction between the way in which single instances are encoded and the way generalizations are captured. Before describing this interaction, and how it is used to understand hierarchies, several major issues must be discussed.

The first of these issues is the identification of the concepts needed to understand hierarchies, and the formalism necessary to present the MERGE scheme. There are many issues involved in representing single hierarchies that can be discussed outside of the MERGE scheme. Similarly, generalization issues that have to do with understanding hierarchically structured objects can be described in isolation. Chapters 4 and 5 discuss issues in representation and generalization, respectively.

This chapter deals with the basic principles of hierarchy understanding and presents the requisite formalism and definitions needed to describe representation issues, generalization issues, and the MERGE scheme.

In order to develop a hierarchy understanding system one must first identify exactly what we mean by a hierarchy. One dictionary definition of a ***hierarchy*** is: "a group of persons or things arranged in order of rank, grade, class, etc."[1] There must be a relation between members of a hierarchy that determines this ordering. We call this the ***fundamental relation***, or ***F-rel***, of a hierarchy. We take the F-rel to be a partial ordering, allowing multiple members of a hierarchy to be subservient to the same member. The F-rel of a hierarchy allows nodes to be built into a tree-like structure.

3.2 Fundamental relations and trees

A hierarchical system can be characterized by more than one F-rel. Such a system is called a tangled hierarchy. Because of this possibility, it is necessary to specify what organizational concept is being captured by a particular F-rel. Figure 3-1 shows some sample domains/organizational concepts and their corresponding F-rels. An ***organizational concept*** in a hierarchical domain is a generalized notion of how members of a hierarchy relate to one another.

Although we do not dispute the existence (or validity) of tangled hierarchies, throughout the remainder of this thesis we will usually view a hierarchy as having one predominant decomposition based on a single F-rel. Aside from pragmatic aspects (e.g., single F-rel hierarchical systems are easier to discuss, and write programs to understand), applications of MERGE (e.g., RESEARCHER and CORPORATE-RESEARCHER) need only use single F-rel decompositions to achieve their goal of understanding hierarchies. A MERGE-based system is designed to understand a specific domain. Furthermore, there is likely to be one preferential way of viewing this domain (although, not always). This view, or organizational concept, is the one of most interest and would be the F-rel used by the program. We will show how information captured by multiple F-rel decompositions can be included in a single F-rel representation. (This is done later in this section.)

A hierarchy is built by joining nodes (members of a system) with F-rel links. For example, the IS-A link (see Chapter 2 for a more complete description of IS-A links in semantic nets) serves to indicate that one node is an instance of another node, and hence is subservient to it. We would say that *General_Motors* IS-A *manufacturer*, indicating that manufacturer is higher in the (corporation) IS-A hierarchy than General Motors. Both General Motors and manufacturer are nodes in this hierarchy, the IS-A link connecting them is an instantiation (particular

[1] From Webster's New Twentieth Century Dictionary, second edition.

Domain	Organizational Concept	F-rel
human organizational systems (e.g., government, corporations)	chain-of-command	REPORTS-TO
physical objects (e.g., cars, disc drives)	sub-assemblies	PART-OF
genealogy (e.g., family trees)	birth	CHILD-OF
road/river systems (e.g., interstate highways)	tributaries	FEEDS-INTO
human symbolic systems (e.g., music, books)	units of meaning	PART-OF
taxonomies (e.g., biological taxonomies, subatomic particle classification)	classification	IS-A

These examples show that a hierarchical domain has a particular F-rel (fundamental relation) that forms the backbone of its structure.

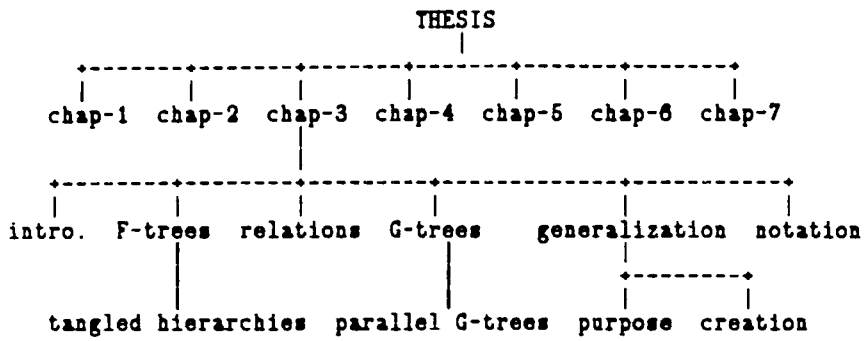
Figure 3-1: F-rels of hierarchical domains.

instance) of an F-rel. PART-OF is another commonly used F-rel. (See [Winograd 72] for examples of this F-rel in the blocks world.) Here the relation denotes that one object is physically included within another. A third example, from the business community, is the REPORTS-TO F-rel. This relation is used by corporations to build an organizational hierarchy.

A few additional definitions are needed before continuing. An F-rel always describes the relation between two nodes in a specific order. That is, if node X is subservient to node Y then the relation reads "X F-rel Y" (e.g., X is PART-OF Y, X REPORTS-TO Y, etc.). The term *F-child* (Fundamental relation child) refers to a specific instance that has an F-rel link to its *F-parent*. We would say that "X is an F-child of Y", or "Y is the F-parent of X".

An *F-tree* (Fundamental relation tree) is simply the hierarchy formed by connecting nodes using F-rel links. These hierarchies are tree-like in appearance

and hence the term F-tree. Such a tree describes a single instance of a hierarchy. Figure 3-2 shows a partial F-tree for this thesis using the PART-OF F-rel. Although other relations may exist within an F-tree (which we then call an *augmented* F-tree) it must use only one F-rel to form its basic structure.



This is a partial picture of the F-tree of this thesis. The F-rel for this example is PART-OF. The levels of this F-tree are: document (thesis), chapter, section, sub-section.

Figure 3-2: Partial thesis F-tree.

Although the F-rel constitutes the main connecting link among nodes in the structure, other relations among arbitrary nodes also exist in any real-world hierarchy. For the purposes of this thesis, a hierarchy will be defined by a single F-rel imposing a partial ordering on the nodes that comprise it along with other relations that augment this structure. For example, channels of communication between members of a corporation other than REPORTS-TO exist, but are considered non-fundamental relations. The F-rel is of primary importance as it determines the structure of the hierarchy.

3.2.1 Tangled hierarchies

The reason why we discuss tangled hierarchies is to show how the information contained in alternate decompositions of a system can be captured by non-fundamental relations. To see how a tangled hierarchy might look, consider the two F-trees shown in Figure 3-3. The F-tree based on the PART-OF F-rel (Figure 3-3(a)) gives a partial decomposition of a *car* according to its physical construction. The same four leaf node components (i.e., the *radio*, the *engine*, the *drive-wheels*, and the *transmission*) also have a decomposition based on their function. The F-rel for the F-tree shown in Figure 3-3(b) is FUNCTIONS-AS-PART-OF. For simplicity, we have not drawn these two F-trees as a single tangled hierarchy. However, they could easily be drawn as such by allowing both F-trees to share the same four leaf nodes.

The functional F-tree contains information that could be partially captured by overlaying relations among various nodes in the PART-OF F-tree. We demonstrate

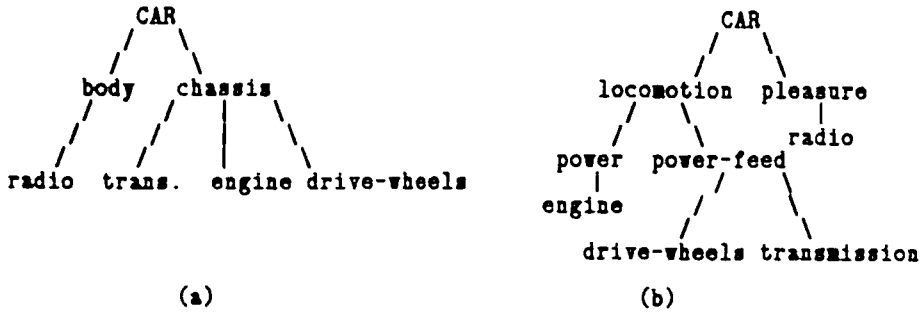


Diagram (a) shows the component decomposition of a CAR using the PART-OF F-rel. The *transmission* (trans), *engine* and *drive-wheels* are PART-OF the *chassis*; while the *radio* is PART-OF the *body*. The F-rel implicit in diagram (b) is FUNCTIONS-AS-PART-OF. That is, the *engine* is a functional part of the *locomotion* process, the *transmission* and *drive-wheels* are a functional part of the *power-feed* system and the *radio* is used for pleasure.

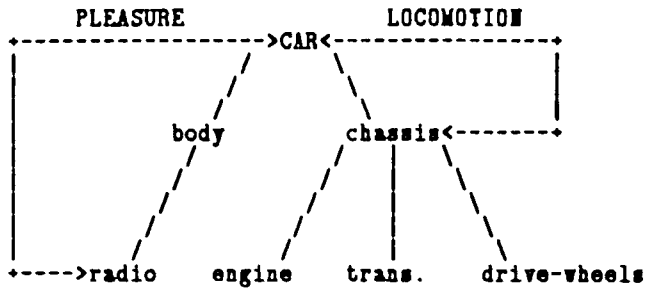
Figure 3-3: Two automobile F-trees.

this in the next section. However, this would be counterproductive if it were necessary to describe a hierarchy according to how it works as opposed to how it is structured. Thus, if an intelligent information system were designed to understand both the composition and function of automobiles, it would need to be able to represent and generalize about these two types of F-trees simultaneously. As we will present it, a single MERGE-based system is designed to work with a single F-rel. One must choose which F-rel to use for a particular application. However, one could simply use two separate MERGE-based systems if it were desirable to understand hierarchies according to both function and part decompositions, for example.

3.3 Other relations

Relations other than the fundamental one are important in forming a complete representation of a hierarchy. They are used in MERGE to capture what other researchers would try to do with tangled hierarchies. The degree to which non-fundamental relations affect a representation depends on the domain of study. In some domains (corporate hierarchies, for example) most of the hierarchy's meaning is carried by the basic F-tree (based on the REPORTS-TO F-rel). To be sure, there are many other relations in a corporate structure, but when companies "chart" themselves they usually don't include too many of these, indicating that they are of secondary importance. In other domains, non-fundamental relations can be very important. For example, modular programming stresses that subroutines should be hierarchically organized as far as flow-of-control goes, but they often pass data back and forth through more complex pathways. These relations between subroutines are crucial to a program's working and must be included in any representation of its structure.

How can information that might come from these non-fundamental relations be incorporated into an object's representation? In particular, our goal is to capture the relational data that would otherwise be encoded by other F-trees or by arbitrary relations among nodes in a single F-tree. For example, the functional F-rel information encoded in Figure 3-3(b) can be overlayed in the PARTS-OF F-tree (Figure 3-3(a)) by using additional relations. Figure 3-4 demonstrates how relations can be used to supplement the basic component F-tree of Figure 3-3(a). Here, most of the data from the functional F-tree has been incorporated into the component F-tree, forming an *augmented F-tree*.



The simple PART-OF F-tree from Figure 3-3 (a) has been augmented by relations that appeared as F-rel links in the F-tree in Figure 3-3(b).

Figure 3-4: F-tree with relations.

Some relations derived from the F-rel links in the functional F-tree have been used in Figure 3-4. We have represented them as having meanings more specific than FUNCTIONS-AS-PART-OF, each relation expressing the exact nature of each components function (e.g., the *radio* is used for PLEASURE). Relations are used in MERGE to afford such specificity for representing complex objects. Although the *chassis* does not explicitly appear in Figure 3-3(b), the LOCOMOTION function uniquely involves all the components of the *chassis* and therefore we have placed the LOCOMOTION function between the *car* and the *chassis*. As shown, non-fundamental relations can occur among arbitrary nodes in an F-tree -- the nodes involved in a relation need not be at the same level in the F-tree.

Figure 3-5 illustrates some of the possible relations that can be used to augment F-trees in various domains. Both classes of relations and specific examples are shown.

Each domain/area has a particular set of relations. In general, it is much harder to uncover all possible relations than it is to determine F-rels for a domain. It is quite clear that human organizational systems (governments, corporations, etc.) devise their hierarchical structure based upon chain-of-command (i.e., the REPORTS-TO F-rel). However, the intricacies of other relations existing within a bureaucracy are virtually impossible to spell out. Even if one could formulate a closed set of possible relations that exist among members of such a hierarchy (e.g., advises, assists, is-located-near, doesn't-speak-to, etc.), it would be astonishing if

Domain (F-rel)	Relation Classes	Examples
physical objects (PART-OF)	positional purposes	left, above supports
corporate hierarchies (REPORTS-TO)	personnel physical	advises, is-friends-with in-same-region
road systems (FEEDS-INTO)	routings	short-cut, intersects
structured programming (CALLED-FROM)	idiosyncrasies data flow	goto statement common areas, side effects

Shown are some sample relations for various hierarchical domains that can be used to augment an F-tree in a particular domain.

Figure 3-5: Other relations of hierarchical domains.

every instance of these relations could be identified. If every instance could be captured, we claim one would have a complete representation of the hierarchy, but pragmatics dictates that this is unlikely for real-world hierarchies. Relations other than the F-rel act as a refinement to the hierarchy's representation, but usually cannot completely capture all the information the hierarchy embodies, because they cannot all be identified.

The "usually" qualifier in the above sentence was put in so that some artificial hierarchies would not be excluded -- hierarchies that are created in the abstract so that they may be completely understood in terms of a fixed number of relations. For example, biological taxonomies classify all specifies of animals according to a seven level F-tree. Each level is named and the sequence is: kingdom, phylum, class, order, family, genus, and species. There exists a set of rules (periodically updated) that biologists follow in order to make classifications. Therefore, relations between nodes in this particular F-tree should be enumerable according to this set of rules. This is a narrow example of a domain (only one full F-tree is used) but other artificial domains with many F-trees have similar properties.

Writings of various types are hierarchical in form. There also exist other relations among pieces of text. For example, this sentence (which can be viewed as a node in the PART-OF F-tree of this thesis) refers to Section 3.2 where a partial F-tree for this thesis is presented. It is possible to enumerate all such relations among pieces of text in a document because they have been put there intentionally (e.g., the previous sentence is an intentional reference to Section 3.2). Thus, writing is a

domain in which artificial hierarchies have an identifiable set of non-fundamental relations (excluding nebulous relations such as motifs, style, etc.).

The number of relations employed in augmenting an F-tree varies greatly from domain to domain and from instance to instance. In general, the more evolution a hierarchy undergoes, the more supplemental relations it will acquire. For example, governmental structures often start life in the form prescribed by a constitution. As legislation is written that creates new checks and balances, as well as new departments, the hierarchy becomes more complex. Although the F-rel remains the same (REPORTS-TO), more relations are added, hence creating a bureaucracy. Similarly, large computer programs are usually built in a clean, modular form. But as new features are added, it is common practice to end up with the need for arbitrary modules to pass data back and forth. Each code patch of this form adds another relation to the F-tree. At some point, the number of added non-fundamental relations can get out of control. This may be a signal that the hierarchy has to be restructured. (Programmers are quite familiar with this phenomena when working on large systems.)

Although relations are secondary in importance to F-rel links, they still play a major role in a MERGE-based system. Aside from augmenting an instance hierarchy's meaning, they are also used in generalization. Relations, in addition to F-children, are used to determine which nodes should correspond when matching F-trees against one another. They are an integral part of the generalization process. This means that they must be compared and their commonalities and/or differences recognized. Because relations are usually more complex than F-rels, they need special representation schemes in order to encode them and generalize about them. This issue will be explored further in Chapters 4 and 5.

3.4 Generalization principles

Individual instances of hierarchies can be represented by augmented F-trees. However, our goal of building a hierarchy understanding system requires that instances be generalized and indexed into a unified knowledge base in order to capture the similarities among examples. That is, a generalization-based memory (GBM) of the sort used in earlier work [Lebowitz 80; Lebowitz 83c] is needed. A MERGE-based system uses two types of hierarchies, F-trees and generalization hierarchies. A generalization hierarchy ties together individual F-trees of whatever domain is under investigation into a hierarchy of generalized concepts. In doing so, the instance F-trees are modified to emphasize how they vary from the generalized F-trees created within the system. The generalizations created, of course, depend on what data the instance F-trees contain. Thus, F-trees become inexorably linked with the generalization hierarchy.

We call a generalization hierarchy a *G-tree* (Generalization tree). The name given

to it is indicative of the structural similarity it has to an F-tree. In fact, it is an F-tree with the special F-rel, VARIANT-OF. The term *variant-of* has essentially the same meaning as *is-a* but is used by us specifically to refer to links in G-trees.

In this section we will motivate the reasons for generalizing and explain roughly how the process is carried out. The details of generalization in MERGE are discussed in the next section and in Chapters 5 and 6.

3.4.1 The purpose of generalizing

Humans learn in many ways. When they are presented with a large number of examples of a phenomena, probably the most useful form of learning is generalization. We distinguish three purposes behind generalizing (for both humans and intelligent information systems). Generalizations are used: 1- to categorize instances into logically organized groups, 2- to emphasize similarities and differences among instances, 3- to help in understanding future input through the use of prototypes.

Organizing instances into categories is an obvious application for generalization. People constantly do this when confronted with a domain that has many instances. For example, household furniture can be grouped according to what room it belongs in, what function it serves, how much it costs, etc. We create concepts such as: "low-cost table and chair sets", and "expensive antique desks". Without such concepts it would be difficult to imagine how we could walk through a furniture show room and not be totally confused.

These concepts can be built into a tree-like structure that acts as a discrimination network. If chairs are one category of furniture then it may be sub-categorized into expensive chairs and inexpensive chairs. The expensive chairs category may be further subdivided into those with arms and those without arms. This branching continues until individual instances of chairs are reached at the leaf nodes of this generalization tree.

The second purpose for generalizing is to show how similar or different instances (or generalized concepts) might be from one another. The use of *inheritance* in a generalization hierarchy makes clear both what instances have in common and what they don't. Inheritance is the process of acquiring data that is absent (from the physical representation) in an instance but present in the generalized concept of which the instance is a variant. The lowest common ancestor of nodes in a generalization tree contains the information common to all its variants (instances). The differences between two instances are found by juxtaposing these two leaves of the G-tree. Since inheritance will factor out any similar data only the differences remain stored in these nodes.

Finally, generalized concepts serve as prototypes. A prototype is simply an old

instance or, more likely, a generalized concept that is used as a comparison standard for new instances. Prototypes are useful as a model with which previously unencountered instances can be compared. If new instances are incomplete or ambiguous, the previously established concepts can be used to fill in missing data or establish a norm for ambiguous data. Thus, generalization can be helpful in understanding future input. Once a new instance is incorporated into memory, inheritance is used to factor out commonalities and allow the differences to stand out.

3.4.2 Creating generalizations

In the real world, learning takes place incrementally over time. Humans are exposed to instances of phenomena in pieces -- not all at once. A generalization scheme must be able to work in an incremental mode if it hopes to understand hierarchies in a real-time situation. Incremental generalization is necessary in order to provide an ongoing representation of what the system knows. The alternative would be to wait until all (or several) instances were known before generalizing about them. This may not be acceptable performance in a real-time intelligent information system.

The basic idea of incremental generalization is that a single instance is incorporated into an existing knowledge base by making minor updates. In MERGE, this means that each F-tree is incorporated into one or more G-trees by making it a variant of some existing generalization or creating a new one. Addition, subtraction, and substitution operations are used to augment the standard inheritance process, when necessary.

Usually a new F-tree is incorporated into the existing knowledge base when its representation is complete. However, depending on the domain in which MERGE is applied, it is possible to start incorporating an instance into G-trees before its representation is completely formed. When a partial F-tree is created, from either a bottom-up or top-down description, it can be incorporated into a G-tree. Each G-tree categorizes a different type of F-tree. Thus, the entire F-tree need not be described before a generalization can be made. For example, when reading a patent abstract about a disc drive (as RESEARCHER does), if the drive motor assembly is described first, its F-tree can be incorporated into the G-tree that categorizes motors before the entire patent abstract is read. In this way, a system can avoid having to process too much at once while locating where in its knowledge structures the description of the current instance fits in. Knowing this location early on during the knowledge acquisition process is useful because the information stored in the concepts around this point may help in disambiguating the rest of the description.

In addition to disambiguation, information missing in new instance representations can be filled in in accordance with the prototypes in the knowledge base. When a

new instance hierarchy is found to closely match some generalized concept, data that is absent in this instance, but present in the generalization, can be assumed. The circumstances governing when this should take place are discussed in Chapter 5.

3.5 Generalization trees

We have described the principles behind generalization but not the details. G-trees are the second major data type in MERGE, F-trees being the other. In this section we introduce the concepts needed to create and manipulate G-trees. The previous section only mentioned these concepts which include inheritance, addition, subtraction, substitution, and the use of multiple G-trees. Here we clarify them and exemplify how MERGE represents and processes G-trees.

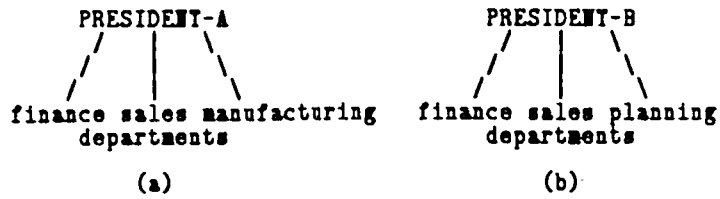
3.5.1 Inheritance and modifications to it

F-tree representations are incorporated into a G-tree structure mainly through the inheritance operation and variations of it. Members of one instance F-tree that are the same as members of a different F-tree are factored out and stored in generalized F-trees (i.e., generalized concepts). These common elements are then removed from the representations of the instance F-trees so that they do not explicitly appear there. Instead, they are inherited from the generalized F-tree, thus accentuating what the instances have in common and what their differences are.

Consider the simple corporate F-trees shown in Figure 3-6. The F-rel of these trees is REPORTS-TO. If they were the only two instances of a presidential structure they would generalize to be the F-tree in Figure 3-7. This type of generalization, where only common elements are recognized, is called a *conjunctive generalization*. By inspection of this generalized concept of a president (*president-#*), it is immediately clear what its variants (*president-a* and *president-b*) have in common. If one looks at what remains (i.e., is not inherited) in the variants of this generalized concept, then the differences between *president-a* and *president-b* are apparent.

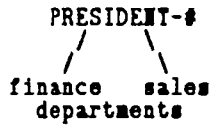
The complete G-tree, using this generalization, is shown in Figure 3-8. Here, inheritance has been used so that *president-a* and *president-b* will get copies of both the *finance* department and the *sales* department representations. Inheritance occurs along the VARIANT-OF links in the G-tree (indicated by the *v's*).

Closer inspection of this miniature knowledge structure reveals that what has been represented is a concept of a president (*president-#*) who has control of a *finance* department and a *sales* department. There are also two variations of this stereotypical president: one (*president-a*) adds a *manufacturing* department to its span-of-control (see Section 2.4), while the other adds a *planning* department.



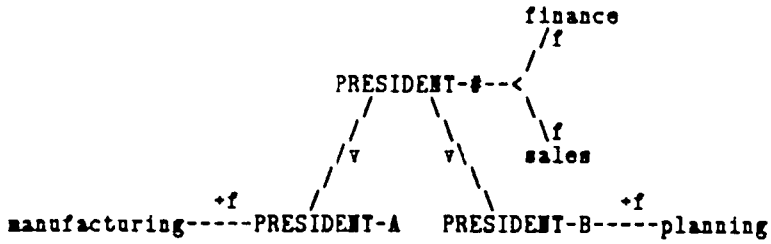
Two simple corporate F-trees, based on the REPORTS-TO F-rel, are shown. Note that both *president-a* and *president-b* control *finance* and *sales* departments and that they differ in that *president-a* also has a *manufacturing* department while *president-b* oversees a *planning* department.

Figure 3-6: Simple corporate F-trees.



This F-tree represents the result of generalizing *president-a* and *president-b* shown in Figure 3-6.

Figure 3-7: Simple generalized president.

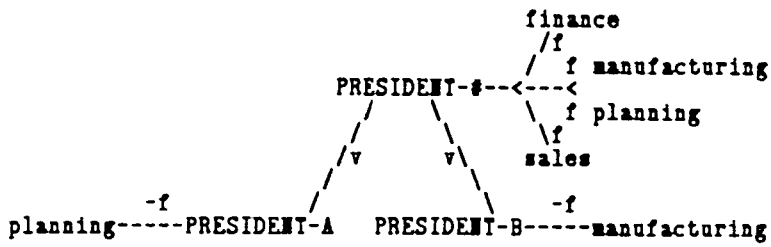


The F-trees from Figure 3-6 are indexed as VARIANTS-OF *president-#*. The *f* symbols represent F-rel links while the *v's* indicate the G-tree pointers (i.e. VARIANT-OF links). A G-tree is an F-tree that has VARIANT-OF as its F-rel. *President-a* and *president-b* inherit the representation of both the *sales* department and *finance* department from the generalized concept of a president (*president-#*). The "+" symbols indicate that this is an added F-rel link.

Figure 3-8: G-tree using added-inheritance.

The concept of *added-inheritance* is used to distinguish specializations of a generalized object. It correlates to the phrase "X is just like Y only it also has a Z". The converse situation, "Y is just like X only it's missing Z", is also possible using the *subtraction* operation. If, for example, the generalization that a president has a *finance* department, a *sales* department, a *planning* department, and a *manufacturing* department had been made (which is not a conjunctive generalization) then subtraction could be used to modify inheritance so that certain F-children are deleted after the inheritance. We call this deletion of inherited F-

children *deleted-inheritance*. Figure 3-9 shows how deleted-inheritance is used to represent the same information as Figure 3-8 does for added-inheritance.



Assuming that the generalization mentioned in the text is adopted, then the G-tree will appear as shown in this diagram. Notice that the subtraction operation (-) is needed to permit deleted-inheritance.

Figure 3-9: G-tree using deleted-inheritance.

In Figure 3-9 we have assumed that the generalized concept of a president (*president-#*) is one that has four F-children. It has been formed by the union of the F-children from *president-a* and *president-b*, as opposed to the intersection of the same data, which appears in Figure 3-8.

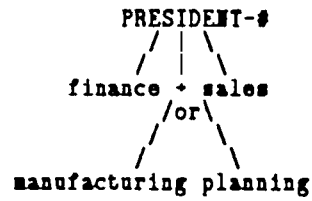
Although the representations (i.e., the G-tree and F-trees) shown in Figure 3-8 and Figure 3-9 are different in form, they encode exactly the same information. The choice of one representation over another is dependent on the current state of the knowledge base and what new instances are to be incorporated into it. Generally speaking, one (a MERGE-based system) would choose the representation that minimizes the total size of the representation, while maintaining cognitive accuracy. For example, if a knowledge base contains many instances of tables with four legs and a new instance of a table with three legs is presented, the best generalization would include four legs in its encoding. Deleted-inheritance (i.e., the subtraction operation) would be used to capture the fact that the new instance has one less leg. The question of which generalization to make is taken up further in Chapter 5.

A third operation, *substitution*, combines both subtraction and addition to allow for the replacement of one member in a generalized F-tree by another, more specific, element from an instance F-tree. It is used in representing concepts such as "X is just like Y but it has P instead of Q." Using the corporate president examples in Figure 3-6, one could represent *president-b* as a VARIANT-OF *president-a* with the substitution of the *planning* department for the *manufacturing* department.

The substitution operation must be used sparingly. The purpose of using the substitution operation is to emphasize that its two arguments (old node and new node) are similar in structure and should be thought of as variants of the same concept. Limiting its usage helps prevent unreasonable knowledge representations such as "a telephone is just like a calculator, only you subtract everything but the

key-pad and add a handset, cord, bell, etc." from being created. A guideline for restricting its usage might be to only allow a small percentage of a concept's F-children to be substituted for in any one variant of that concept. Thus, if a generalized F-tree had four F-children (and we arbitrarily take this percentage to be 25%) then only one F-child would be permitted to have a substitution, otherwise a new generalization would have to be built.

It is also possible to make a *disjunctive generalization* about *president-a* and *president-b* and produce an F-tree that has the structure shown in Figure 3-10. Actually, this generalization is a combination of conjunction and disjunction. The two instance examples (*president-a* and *president-b*) support the fact that a generalized president has both a *finance* department and a *sales* department. It also has either a *manufacturing* department or a *planning* department, but not both. Disjunctive generalization, alone, is simply the generalization formed by taking the logical disjunctions of the F-children in the variant nodes of a concept. (The *president-#* F-tree shown in Figure 3-9 is an example of a purely disjunctive generalization.) Although the use of conjunctive and disjunctive generalizations simultaneously is a powerful technique, we will continue to only use the former type for our purposes. However, in Chapter 5 we will discuss disjunctive generalization further, and why we should or should not use it.



If disjunctions are allowed, the generalized president (*president-#*) can be represented as having control of a *finance* department, a *sales* department, and either a *manufacturing* department or a *planning* department, but not both.

Figure 3-10: Disjunctive/conjunctive generalization tree.

3.5.2 Parallel generalization trees

In an intelligent system that understands hierarchies, it is desirable to have the system generalize about all of the sub-hierarchies that exist within the whole. Each sub-hierarchy of an object is itself an object which might appear in other contexts (e.g., the drive-motor assembly of a disc drive). The MERGE scheme builds many G-trees simultaneously, and consequently learns about objects that are part of the top-level object in the domain as well as the top-level object itself (e.g., it would learn about drive-motor assemblies as well as disc drives). Thus, the range of knowledge the system can handle is broadened, which is a desirable effect in almost any application.

Parallel G-trees coexist in such a manner as to organize F-tree data at different

levels in the F-rel hierarchies. These G-trees do not have any nodes in common with each other; they are independent indices into the F-tree forest. Usually there are as many parallel G-trees as there are unique members in the F-trees being generalized about. Of course, not all F-trees in a given domain have the same number of unique nodes, so this should be treated as a general notion. (We will discuss how F-trees with varying depths and numbers of nodes can be generalized about in Chapters 5 and 6.)

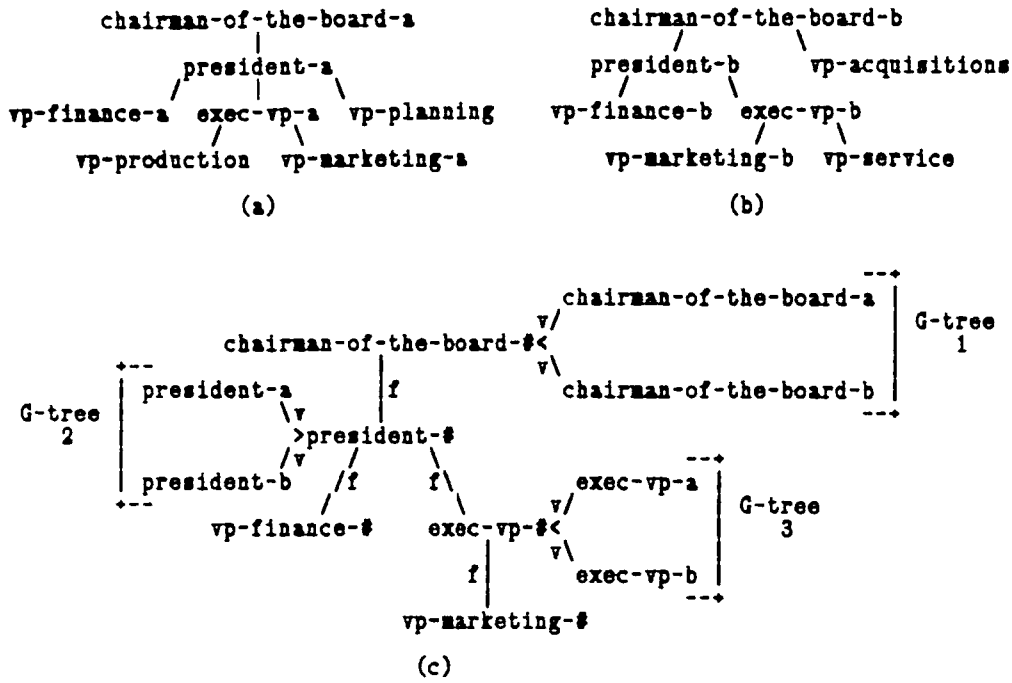
Parallel G-trees allow MERGE to extend its knowledge base without doing much additional work. In the process of comparing two hierarchical F-trees, their sub-hierarchies (sub-F-trees) must be compared. All that needs to be done in order to build multiple G-trees is to save the results of this comparison in the form of a generalized concept. Thus, MERGE learns about objects in whatever domain it is employed in, as well as any sub-objects (hierarchies) that exist within the complete instance hierarchies it processes. Each G-tree, therefore, is a knowledge structure that classifies a different kind of object.

To see how parallel G-trees come about, consider upper level corporate management hierarchies. A G-tree that serves to categorize F-trees in this domain would, of course, store generalizations about corporate structures in total. If all the sample corporations had a chairman of the board at the head of the company followed by a president, followed by an executive vice president, then the generalized corporation would also have a chairman, president, and executive vice president.

There is, however, more information that can be extracted from a comparison of instance hierarchies. Specifically, one can make generalizations about each level in the F-trees. In the upper level corporate management domain, this means that representations of a generalized president, executive vice president, and other officers can be made. Each generalization exists in parallel with the others. Figure 3-11 illustrates these points. Here, the complete F-tree for *chairman-of-the-board-#* is shown along with three G-trees that have been created from a comparison of *chairman-of-the-board-a* and *chairman-of-the-board-b*. The details of added-inheritance (the only operation needed for this representation) are not shown. Note that generalized concepts of the *chairman-of-the-board-#*, *president-#*, and *exec-vp-#* have been created. Also consider that each of the three G-trees classifies different nodes in the instance F-trees. The details of parallel G-trees are discussed fully in Chapter 5.

3.6 Notational formalism

Up to this point we have been diagramming very simple hierarchies to be used as examples to describe some fundamental principles. Real-world instances, as well as more complex hypothetical examples, would be difficult to diagram in this way (i.e.,



Trees (a) and (b) represent two hypothetical corporate structures (F-trees). The generalization of these two structures appears in (c) as *chairman-of-the-board-#*. Note that this is a somewhat different diagram than appears in Figure 3-8 and Figure 3-9. Here, the generalized F-tree is shown as the main structure with VARIANT links off to the sides. Three G-trees (1,2,3) are shown, they serve to classify the chairman, presidents, and executive vice-presidents, respectively.

Figure 3-11: Parallel generalization trees.

as inverted tree structures). In particular, multi-level F-trees embedded in multi-level G-trees are difficult to draw and comprehend. To alleviate this problem and provide a convenient way to proceed with further discussions of the issues of generalization, representation, and their interaction, we introduce a notational formalism.

In order to facilitate our presentation of F-trees, G-trees, and their interrelations, we need a concise notation. The essential facts which include: given any node, what are its immediate descendants in the F-tree (i.e., F-children), and what are its immediate G-tree variants (i.e., instances and/or other generalizations) must be made apparent by a good notational scheme. Furthermore, such a scheme should be flexible enough to allow us to add new operators into it.

Figure 3-12 introduces the basic notational scheme we have chosen to accomplish the goals outlined above. The name given to identifiers can be abstract single letter codes, abbreviations, or more descriptive names. To put the abstract structure described in this figure into perspective, assume that the F-rel in this example is PART-OF. Then, this structure defines two objects, a *computer* and a

Encoding	Meaning
COMPUTER: disc-drive, cpu	the *computer* has F-children *disc-drive* and *cpu*.
DISC-DRIVE: disc, >floppy-disc-drive	the *disc-drive* has an F-child, *disc*, and G-tree variant, *floppy-disc-drive*.
DISC: magnetic-coating	the *disc* has an F-child, *magnetic-coating*.
CPU:	the *cpu* has no F-children nor G-tree variants.
FLOPPY-DISC-DRIVE: +drive-door	the *floppy-disc-drive* has an F-child, *drive-door*, in addition to the ones it inherits from *disc-drive*.

Each node in an F-tree or G-tree that has children is represented by its name followed by a list of its descendants, a colon delineates the parent from the children. G-tree variants are prefixed by a ">" to distinguish them from the F-children. A "+" symbol prefixed before an F-tree descendant indicates that it is an *added* F-child (in addition to those that it inherits from its parent(s)). The F-rel of the hierarchy is implicit here and must be stated outside the context of the encoding.

Figure 3-12: Notation for a unified memory structure.

floppy-disc-drive. The *computer* has parts *disc-drive* and *cpu*; the *disc-drive* in turn has a part, *disc*, while the *cpu* has no parts. Furthermore, the *disc* has a *magnetic-coating* as a part. The second object, *floppy-disc-drive*, is actually a VARIANT-OF the *disc-drive*. Thus, it inherits all the parts (F-children) that *disc-drive* contains (i.e., the *disc*). It also has an additional part that the *disc-drive* does not have, namely the *drive-door*. In total, the *floppy-disc-drive* has a *disc* and a *drive-door* as immediate parts.

We call this combination of F-trees and G-trees a *unified memory structure*. It is a merging of F-trees and G-trees into a single representation. All the nodes of importance (i.e., that have either F-tree or G-tree descendants) in this structure are enumerated. Each node listed is in exactly one F-tree (although copies of a node can be inherited), and possibly one or more G-trees (as indicated by the ">" prefix).

The subtraction operator, "−", is used in the same way as the addition operator (+). It specifies that an F-child is to be deleted from the list of inherited F-children. Substitution is similar to doing both a subtraction of the original F-child followed by an addition of the new F-child. Thus, the notation used to symbolize substitution, "−+", has been chosen to make this property clear.

To gain an appreciation of the economy that this notation affords, consider the

representation of the president G-tree described in the previous section. Figure 3-13 shows how the unified memory structure in Figure 3-8 is encoded using our compact notational scheme.

PRESIDENT-#: finance, sales, >president-a, >president-b
PRESIDENT-A: +manufacturing
PRESIDENT-B: +planning

This is the same unified memory structure as shown in Figure 3-8 using our notational scheme.

Figure 3-13: Unified memory structure using added-inheritance.

The unified memory structure using deleted-inheritance (shown in Figure 3-9) has an equivalent representation using this formalism, and is diagrammed in Figure 3-14. Notice that this encoding requires a few more symbols to represent compared to the G-tree using added-inheritance. However, this discrepancy disappears when more than two objects are generalized, or the objects are more complex. Consider the case of having nine identical F-trees, each composed of several components, and a tenth object tree that lacks just one of these parts. Using deleted-inheritance, the missing component can be accounted for with a single additional symbol in the representation. Added-inheritance requires that the nine identical objects all have the "missing" part added, and that the generalized concept (two-level F-tree) not have it in its representation. Therefore, the use of subtraction can prove to be economical and often cognitively accurate.

PRESIDENT-#: finance, sales, manufacturing, planning,
>president-a, >president-b
PRESIDENT-A: -planning
PRESIDENT-B: -manufacturing

Deleted-inheritance can also be represented in this notational formalism. The "-" symbol indicates the subtraction property.

Figure 3-14: Unified memory structure using deleted-inheritance.

The substitution operator can be used to represent *president-a* as a VARIANT-OF *president-b*. Figure 3-15 shows this encoding.

The way in which a unified memory structure is written makes it easy to identify F-children and G-tree variants. However, *F-parents* and parents of nodes in the G-tree can also be seen in this structure. Thus, all four pointer types in MERGE are encoded (in some manner). Cognitively, some links are stronger than others in the sense that a person can enumerate them more readily. In particular, VARIANT-OF links are more easily enumerated than are VARIANT links (the reverse of

PRESIDENT-A: finance, sales, manufacturing,
>president-b
PRESIDENT-B: manufacturing-+planning

President-b is shown as a VARIANT-OF *president-a*. The substitution operator ("-+") is used to indicate that the *manufacturing* department is dropped and the *planning* department is added in its place.

Figure 3-15: Example of the substitution operation.

VARIANT-OF links). For example, one can tell that an object is a piece of furniture faster than he can list all pieces of a certain type of furniture. (This discussion is taken up further in the next chapter.)

Relations are included in this notational system by specifying them separately using standard functional form. Thus, the NEAR relations in Figure 3-4 would be: NEAR(*transmission,drive-wheels*), NEAR(*engine,transmission*), etc. Although these relations are binary, n-ary relations are represented in a similar manner. In Chapter 4, representations of relations are discussed further.

3.7 Summary

The most important factor in hierarchy representation is the F-rel. An F-tree is formed by using F-rel links to connect individual nodes in a partial ordering. Although such a structure contains most of the information about a hierarchy, it can be supplemented through the use of additional relations among arbitrary nodes

A generalization tree (G-tree) is an F-tree based on the VARIANT-OF F-rel. It is used to organize instance F-trees and other generalizations in memory. Through the operation of inheritance, modified by addition, subtraction, and substitution, F-trees are incorporated into G-trees. Several G-trees exist in parallel, each one serving to categorize concepts and instances about a unique object in the instance F-trees

The G-trees depend on the F-trees for their structure. Furthermore, the F-trees are modified by the existence of the G-trees. This integration forms the basis for the MERGE scheme of hierarchy understanding. That is, there is a mutually beneficial interaction between these two types of knowledge structures making them appear as a single entity.

The complexity of these representations makes necessary the adoption of a concise notational formalism. The scheme presented is extensible and permits the investigation of the issues relating to representation, generalization, and their integration. The representation of F-trees and G-trees concurrently we call a unified memory structure.

A single F-tree is represented using a frame-based formalism. Each node in a hierarchy is encoded as a memette which stores information about the node's parent in the F-tree, children in the F-tree (i.e., its parts), parent in the G-tree, structure-independent and structure-dependent information. Structure-independent information is applicable only to the immediate memette; the properties of an object is an example of it. Structure-dependent information is that which references multiple nodes in the F-tree; it is tantamount to non-fundamental relations among arbitrary memettes. The use of non-fundamental relations is variable from domain to domain. In complex domains, it is desirable to have a primitive-based canonical scheme for encoding relations. The scheme that RESEARCHER employs is described as an example of such a scheme.

4. Representation Issues

4.1 Introduction

The MERGE scheme of hierarchy understanding is designed to combine representation and generalization in an integrated fashion. However, it is possible to distinguish certain issues as representational and others as relating to generalization. This chapter discusses representation issues, specifically those that have relevance to the MERGE scheme. Chapter 5 does the same for issues in generalization.

As a practical matter, we need an operational definition of the term "representation". The entire knowledge base built by the use of MERGE is a representation, in some sense. It includes representations of single instance hierarchies as well as generalized hierarchies. For the purposes of this chapter and the next one, we will consider *representation* to be only what is necessary to capture the information contained in a single instance hierarchy (i.e., encoding), and *generalization* to be whatever is involved in comparing and joining together multiple instance hierarchies into an intelligent knowledge structure (i.e., processing). Representation deals with F-trees while generalization deals with G-trees. Of course, the MERGE scheme emphasizes that representation and generalization must be intimately linked together in order to achieve hierarchy understanding.

There are four areas of particular importance in representing hierarchies: 1- selecting a representational formalism and determining what data it should capture, 2- deciding how levels in an F-tree are to be structured, 3- examining the nature of non-fundamental relations used to augment an F-tree, 4- using information other than F-rels and non-fundamental relations.

The first of these issues, choosing a representational formalism, pertains to any knowledge-based system. Since MERGE is designed to represent complex

hierarchies, it has particular requirements that must be met by whatever formalism is picked. These requirements will be discussed in the next section. F-trees are the means of encoding hierarchies in MERGE. Thus, another important issue in representation is how the levels in an F-tree are structured (i.e., how an F-rel decomposes a hierarchy). A complete representation of a real-world hierarchical system requires both relations that augment an F-tree and miscellaneous data that captures properties of members of the system. Therefore, the third and fourth areas of interest in representing hierarchies are how to encode these types of information.

We will present each of these areas in the following four sections. Problems and issues common to all hierarchical domains will be discussed. Section 4.6 describes the physical object representation scheme used in RESEARCHER. We present this as an example of a sophisticated relation scheme that demonstrates solutions to many of the problems discussed. It should be noted that in this chapter and the next, issues of relevance to an *ideal* MERGE-based system will be discussed. We are not describing how our implementations work, which is done in Chapter 6. CORPORATE-RESEARCHER and RESEARCHER are somewhat less than ideal systems; however they embody the essential ideas of the MERGE scheme.

4.2 F-tree frame formalism

As described in Chapter 2, there are two basic representation formalisms that dominate work in AI. Both semantic networks and frames offer enough expressive power to encode an F-tree structure. However, we have chosen a frame-based approach to represent hierarchies.

The goals driving our choice of a basic representational formalism are relatively straightforward. Hierarchies of arbitrary depth must be easily represented. The capability to capture relations among nodes and property data for any one node has to be present. Provisions must exist for building these single instance representations into G-trees. Finally, it is desirable for the representational formalism to closely correspond with models of human cognition (specifically, the "chunking" theory of memory).

At first glance, semantic networks seem to be a logical choice. Their node-arc formalism maps well into a hierarchy's representation, without having the added baggage of a frame structure. F-trees could be encoded simply by using the hierarchy's F-rel as the links connecting each member of the hierarchy to its superior node. Unfortunately, there are several drawbacks to using a semantic net-based representation scheme. The use of a frame-based system is preferable for both cognitive and pragmatic reasons.

The reasons for preferring frames over semantic nets have already been examined in Chapter 2. We summarize them here.

Semantic nets do not easily allow for grouping of information. Properties or features associated with a particular node appear no different than members of the hierarchy, making the programmer's job more difficult. Frames permit the location of property data within the object's (node's) individual representation. Another problem with the use of semantic networks is their inability to capture the concept of knowledge "chunking". People are able to remember larger amounts of information by using a hierarchical memory structure in which each node is of a small size any amount of data can be retained. Frames provide exactly this capability.

Memettes[1] are the name that we have given to the frames in our system. The term is indicative of our intention to represent a small chunk of memory. A memette can represent an entire hierarchy (when it is the root of an F-tree) or it can stand for a single ***unitary*** (indivisible) object. A memette, therefore, represents an object at some level of detail in the hierarchy being encoded. Being able to either differentiate between levels of representation or treat them the same is an important element in hierarchy understanding. This issue will be discussed in detail in the next section and in Chapter 5. But it is another advantage that frame-based schemes offer over semantic network-based approaches.

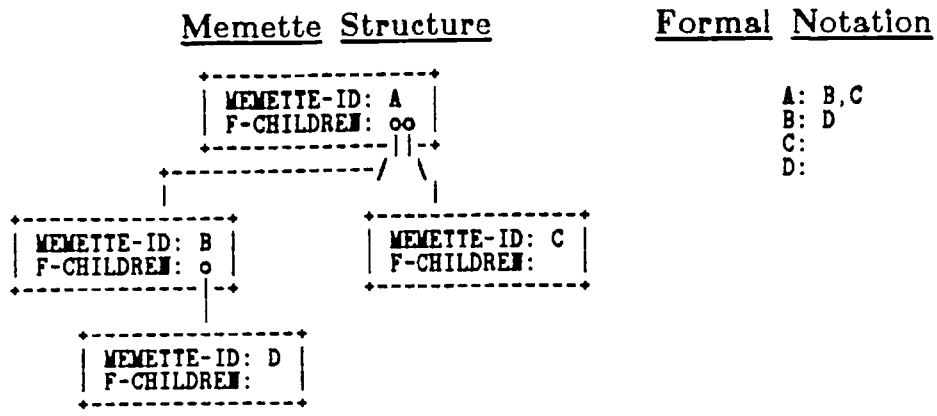
Partitioned semantic networks [Hendrix 79] might be a practical alternative to standard semantic nets. They group data much like frames do and therefore can model the concept of chunking more accurately than un-partitioned semantic networks. Our choice of frames (memettes) over partitioned semantic nets is largely based on our preferential answer to the following question: Is memory (human or otherwise) best modeled by a large collection of links and nodes along with some grouping operation or rather by a system of somewhat larger nodes of information with fewer links between them? Obviously, this is a subtle question that can be debated endlessly. Without answering this, we merely state the we have opted for the frame solution (the later one) and refer the reader to [Schank 82] for further justification.

4.2.1 Memettes

Given that MERGE represents F-trees as memette structures, we next address the issue of what goes into a memette frame. As a reference point, a memette is equivalent to one line in a unified memory structure (i.e., our notational formalism). Figure 4-1 shows a simplified memette structure and its corresponding notational form.

The most important slot in a memette is the one that contains the F-rel pointers. In Figure 4-1 the slot is the only one shown; it has the name F-CHILDREN. It is

[1]Term due to Michael Lebowitz.



On the left side of this diagram a simplified memette structure is shown. The right side shows the equivalent notation for the structure. Note that a memette corresponds to an identifier (e.g., A:) along with pointers to its F-children and any other data that accompanies it.

Figure 4-1: Correspondence of memettes and notation.

simply a list of all the F-children of the current memette. The identifier for a memette is not significant. It may be a descriptive name, but the MERGE scheme only requires that it be unique. However, it is useful to have more descriptive names for examples, as is often used in this thesis.

Memettes can contain other slots. These are divided into two classes; those that contain *structure-dependent* data and those that are filled with *structure-independent* data. By structure-dependent, we mean the information involved refers to nodes in the F-tree. For example, relations among arbitrary nodes are structure-dependent; if the structure of the F-tree changes then any relations involving the effected node(s) would also have to change. Data that is independent of the existence of other nodes in the hierarchy's representation is structure-independent. Properties that are associated with a single memette, such as color or melting point, are examples of structure-independent data.

The distinction between structure-dependent data and structure-independent data is particularly important in generalizing hierarchies. Several difficult problems crop up when structure-dependent data is generalized within the context of F-trees. Our solutions to some of these problems are key elements in distinguishing this research from other work in generalization. The problems and solutions are described in Chapter 5.

Many of the slots that a memette has, other than F-CHILDREN, vary from domain to domain. For example, a physical PROPERTIES slot is useful for representing physical object hierarchies, but not for corporate structures. There are, however, a few other slots that are common to memettes in any domain. These have to do with how memettes are incorporated into G-trees and the reverse F-rel link (i.e., *F-parent*) Figure 4-2 shows the requisite minimum number of slots a

memette must have. A more typical memette frame, essentially the one used in RESEARCHER, is diagrammed in Figure 4-3.

slot name	description of slot fillers
MEMETTE-ID:	unique memette identifier
F-PARENT:	forward (up) F-rel link
F-CHILDREN:	reverse (down) F-rel links
VARIANT-OF:	forward (up) G-tree link
VARIANTS:	reverse (down) G-tree links

Only the required slots that any memette must have are shown here. The names of the F-CHILDREN and F-PARENT slots may be changed for any particular domain so that they are more readable considering the F-rel of the domain. The VARIANTS slot is not a theoretical requirement, but may be practical in many implementations of MERGE.

Figure 4-2: Basic memette slots.

RESEARCHER slot	description
MEMETTE-ID:	
TYPE:	either ***unitary*** for F-tree leaf memettes or ***composite*** otherwise
COMPONENTS:	same as F-CHILDREN
COMPONENT-OF:	same as F-PARENT
VARIANTS:	
VARIANT-OF:	
RELATIONS:	a list of relation records, structure-dependent (see Section 4.4)
PROPERTIES:	structure-independent (see Section 4.5)
PURPOSES:	list of purpose records structure-dependent

Most of the slots used by memettes in RESEARCHER are shown here.

Figure 4-3: Full memette frame.

RESEARCHER and CORPORATE-RESEARCHER use four slots in each memette frame to connect it into the knowledge base. The use of the F-CHILDREN and VARIANT-OF slot seem to make sense from a cognitive perspective. It seems to us that people can usually list the parts of an object and also determine what an object is an instance of (variant of) with relative ease. In addition, it is often easy to say what an object is an F-rel of (i.e., accessing the F-PARENT slot). For example, a leg is a part of a chair, and a vice-president reports to the president.

However, it is usually not easy to list F-rel links for objects that occur in many subparts of complex objects, like screws. It is even more difficult to list all types of furniture, or all corporate officers. This translates to having not so easy access to the VARIANTS slot filler, in a MERGE-based system. In general, slots containing long lists of fillers are more difficult to access (find all fillers) than those with shorter lists.

If cognitive accuracy (with respect to humans) is the only concern in a MERGE-based system, these slots should vary in importance. Perhaps the VARIANTS slot should not exist at all within a memette frame. However, the VARIANTS slot is needed for pragmatic reasons in both RESEARCHER and CORPORATE-RESEARCHER, and probably most any other implementation of MERGE. In an ideal MERGE-based system, about the only use for this slot would be to allow the system to rapidly access instance representations. This would be a useful feature if a system is to act as a database. However, the VARIANTS slot is not needed in order to integrate representation with generalization. The primary integration mechanism, inheritance, along with modification operations (i.e., addition, subtraction, and substitution), need only have the VARIANT-OF slot to work. (It should also be clear that inheritance does not operate along either the F-children link or the F-parent link.) Thus, the VARIANT-OF slot, the F-CHILDREN slot, and the F-PARENT slot are the only theoretical requirements for a memette in MERGE.

4.3 F-rel decompositions

The F-rel decomposition of a hierarchy is an important factor in the MERGE scheme. In this section, we point out the consequences and meaning of such a decomposition. The structure of the F-tree is explored further as well as the significance of levels in the F-tree.

4.3.1 F-tree structure

The concept of *near-decomposability* of hierarchies [Simon 81] indicates that each node of a hierarchy is a more tightly bound unit than are sub-hierarchies in the system. This concept further implies that each subtree of a hierarchy forms a more cohesive structure than do groups of subtrees. From our perspective, near-decomposability (as opposed to complete-decomposability) comes about from two sources, F-rel links and other relations.

F-rel links act as a sort of glue that holds hierarchical systems together. They are the major cause of near-decomposability (i.e., they prevent complete-decomposability), but non-fundamental relations also contribute. There are usually more relations among members of a sub-hierarchy than there are among a group of sub-hierarchies. In other words, relations reference nodes in the hierarchy that are

"close" to each other. This, in effect, adds more bonds to a sub-hierarchy, reinforcing the F-rel structure.

The idea of relative distance within a hierarchy seems fairly straightforward. There are two measures of a hierarchy's structure, *span* and *depth* (see Section 2.4). Span is an indication of how many nodes are subservient to a single parent, while depth measures how many generation (parent-child) relationships exist along a single lineage. Nodes are close if they are either in the same span or in the same lineage, but not too many levels apart.

There is no limit to the complexity of structure that an F-tree can model. The span or depth of a representation can grow arbitrarily large. Of course, there are practical limits in the real-world. When the span of a corporate hierarchy becomes too wide, intermediaries (e.g., vice-presidents, managers, etc.) are interposed between the subordinates and their superior. Thus, the depth of the hierarchy is increased at the expense of the span. Similarly, if the chain-of-command is too lengthy corporate communications can be adversely effected. The solution is to decrease the path length by removing intermediaries and increasing the span-of-control. Corporations try to strike a balance between the span and the depth of their structure [Webber 75]. An ideal MERGE-based system, designed to understand corporate hierarchies, would capture this notion in its generalizations. It may have to explicitly generalize about span and depth to do this. (CORPORATE-RESEARCHER, as yet, makes no attempt to generalize in this way. However, it does represent the interplay between span and depth implicitly -- simply by creating generalizations of entire corporate structures.)

We are interested in the concept of near-decomposability and the measures of span and depth because they indicate how an intelligent information system should process hierarchical systems. Near-decomposability points out that parts of systems are difficult to divide and thus should be processed as if they are autonomous units. A study of span and depth can give insight into the important aspects of instance hierarchies, in a particular domain. These insights can be taken into account when comparing instances and building generalized concepts of them. A simple way of doing this would be to create generalizations based on the number of F-children that a node has as well as the F-children themselves.

Complex hierarchically structured objects are representable using a single F-tree. Each subtree, of this F-tree, is itself a representation of some other hierarchy. According to [Churchman 64], components of systems can be systems themselves. This feature of the F-rel structure is synonymous with the idea of near-decomposability, and can be useful for creating more intelligent information systems. Generalization can be performed on all of the sub-hierarchies within the whole -- not just on the entire object. An understanding system can learn about each unique sub-hierarchy represented in the F-tree, thereby increasing the range of knowledge a system has.

4.3.2 Hierarchical levels

As mentioned previously, the ability to distinguish levels in a hierarchy or its F-tree is important. Here, we investigate the possible benefits of knowing what levels are the most important in understanding a hierarchy. Because G-trees are simply F-trees with the special F-rel of VARIANT-OF, it is appropriate to study the significance of levels in G-trees as well as F-trees.

Although the same F-rel is used to join together any two adjacent levels in a hierarchy, the significance of each level can vary.[1] For example, levels in a corporate hierarchy signify not just the degree of control a member has over subordinates but also determines whether an employee is considered part of management or labor. Going back to the automobile example in the previous chapter, the level of detail might distinguish how parts inventories are kept. Subassemblies of a small size (few constituent parts) might be stored in parts bins, while complex objects (those comprising many parts) may not be stored at all -- they would be built as needed.

Psychologists have given attention to the idea of natural or basic levels of perception in humans ([Rosch et al. 76], for example). Their work demonstrates that there is often some preferred level in a hierarchy, at which understanding of that hierarchy is focused. For example, in a biological taxonomy classifying trees it was found that tree families were the basic level. Thus, in the hierarchical sequence, Tree - Maple - Silver Maple, Maple is the basic level.

As this example indicates, basic levels (or objects) have particular significance for F-trees based on IS-A or VARIANT-OF F-rels (i.e., G-trees in MERGE). The existence of a basic level might imply that the bulk of generalizations made in a domain should center around this level. Increasingly fewer generalizations should be created for other levels in a hierarchy as their distance increases away from the basic level. Thus, if complex physical objects are being generalized, complete items might have little in common (i.e., few generalizations are possible) and it would be senseless to generalize about the nuts and bolts (i.e., extremely low-level detail) that virtually all complex objects have. However, at some level in the hierarchy (the basic level) there could likely be common subassemblies that objects share, and that the generalization process should focus on. The notion that basic levels are rooted in part-wise decompositions of objects is posited in [Hemenway and Tversky 84].

The domain of household appliances provides an example of this phenomena.

---

[1] Simon's work suggests that hierarchies can have different F-rels joining together levels in the same F-tree. We have defined a hierarchy such that it must use the same F-rel throughout. This disparity can be reconciled if the meaning given to the F-rel is sufficiently general so that it encompasses several more specific notions that may be used to join various levels. Refinements to the F-rel's meaning can be accomplished by using non-fundamental relations.

Toasters and refrigerators seem to be be incomparable. All appliances (toasters and refrigerators, included) have screws, nuts, and so forth connecting their major parts. Generalizations based on these low-level parts would be uninteresting. (It is a little like saying "all physical objects are composed of matter".) However, generalizations at higher levels can be interesting. Both toasters and refrigerators (along with a few other appliances, but not all) share a similar electrical system; a thermostatically controlled switch is at the heart of these devices. A generalization system that "thinks" that electrical switches are at a basic level could focus on this fact. Although electric switches may not be a perceptual basic level for most laymen, an engineer might indeed think this way. Generalizing about such subassemblies could prove useful, particularly if one wanted to cannibalize a toaster to fix a refrigerator.

The point to be made here is that identifying the levels of description of an object can have a significant impact on how hierarchies are understood. Although it is possible to make generalizations without a particular focus point much time and effort can be saved by concentrating only on the levels at which interesting generalizations are likely to occur (e.g., the level of electrical assemblies, in the example above). An ideal MERGE-based system would implement this idea. At present, CORPORATE-RESEARCHER and RESEARCHER do not (because we do not know how to accomplish this yet), they make generalizations at all levels in the F-trees. In effect, they assume that all objects are equally important. (Actually, they do distinguish the top-level F-tree nodes from all the rest. This is described in Chapter 6.)

4.4 Non-fundamental relations

Relations within a hierarchy, other than the fundamental ones, can carry a great deal of information. They are composed of a *characteristic* along with a list of arguments. The characteristic is the definition of the relation which may be complex. The arguments are a list of nodes (memettes) in the F-tree that are involved in the relation. Because relations reference memettes, they are a form of structure-dependent data.

The characteristic of a relation is the basic determinant of its meaning. However, there are other factors affecting the use of relations. These include: how arguments should be organized, and how many arguments are needed in a single relation. We discuss these points before describing characteristics.

4.4.1 Some basic observations

Depending on the domain, the characteristics of relations can be complex and obscure a description of the nature of relations. Therefore, we will use an augmented F-tree that employs relations with simple characteristics to exemplify these points.

Figure 4-4 shows a corporate F-tree along with a few relations among its members. The F-rel of REPORTS-TO defines the F-tree and therefore the fundamental relations among the nine memettes, but it does not capture all the information necessary to accurately represent *corporation-a's* structure.

F-tree

CORPORATION-A:	chairman
CHAIRMAN:	president, manufacturing
PRESIDENT:	executive-vp, general-vp
EXECUTIVE-VP:	marketing, research, production
MANUFACTURING:	
GENERAL-VP:	
MARKETING:	
RESEARCH:	
PRODUCTION:	

Relations

SUBSIDIARY(corporation-a, manufacturing)
ADVISES(executive-vp, general-vp)
DIVISION-GROUP(executive-vp, marketing, production)

A nine memette F-tree is represented here along with three relations. The relations cause various memettes to be associated with a characteristic that defines each relation. The relations used in this example are the same as the ones used in CORPORATE-RESEARCHER. The F-rel for this example is REPORTS-TO (the usual one for corporate hierarchies).

Figure 4-4: Augmented corporate F-tree.

The three sample relations demonstrate several facts about the nature of relations. Assume that each relation's characteristic has been defined so that we need not be concerned with it; we can take each definition at face value. (DIVISION-GROUP simply indicates that some members of the corporation comprise a division.)

The first item to note is the number and order of a relation's arguments. In the case of the SUBSIDIARY relation, the first argument indicates the parent corporation while the second argument references the subsidiary company. (In this case, the ***manufacturing*** division is set up as a SUBSIDIARY of *corporation-a*.) The ADVISES characteristic also has two arguments -- the advisee is first followed by the advisor. DIVISION-GROUP can take any number of arguments. The first, however, is unique and points to the lowest common ancestor (corporate superior) that the members of the group report to. The remainder of the arguments form an unordered set that comprises the division group.

Almost all relations are binary in nature. That is, either two objects are related or there is a relation between one object and an arbitrary group of mutually

equivalent objects. We have observed that this is essentially true independent of the domain. Even relations like DIVISION-GROUP are principally a juxtaposition of two objects (the superior and its subordinates). The observation that most relations are binary is helpful in formulating canonical schemes for encoding characteristics, as will be described later. (A common exception to this rule is the *between* relation, and other semantic equivalents, that require three ordered arguments.)

4.4.2 Relation characteristics

In any particular domain, non-fundamental relations are difficult or impossible to completely enumerate. Furthermore, no two domains have exactly the same set of relations, although they may have many in common. An additional concern is the source of relations. That is, in what way are relations made available to an understanding system; through natural language input, hand coded by an expert, or via intuition? These factors make clear that deriving a system of characteristic representations for relations is a formidable task. We briefly discuss these factors in an effort to demonstrate the problems that must be solved in developing a relation characteristic representation scheme. In section 4.6 we present an example of such a scheme developed for RESEARCHER.

Given that relation characteristics are both domain dependent and subject to the input data form, one can only suggest general techniques for developing a scheme. The basic principle on which to base a useful system is ***semantic primitives*** (see Chapter 2). Their use often results in a canonical encoding of varied input data such that different descriptions of the same real-world relation have the same characteristic encoding.

The relations of a particular domain may be so varied that a simple semantic primitive reduction (i.e., using a set of semantic primitives that have no underlying structure) of them is still unwieldy. If the number of simple primitives required to characterize relations in a domain is too large, then little useful functionality is obtained by employing that scheme. Some extension to simple semantic primitives is needed in such domains.

One possible solution is to use a small set of primitives that can be combined to represent a characteristic instead of the usual one to many mapping. Consider the task of developing a system to describe color. If one uses a set of simple primitives (i.e., only one primitive can be used as the description) there would have to be many primitives in order to get an accurate description of any color. The alternative would be to use just a few primitives (e.g., red, blue, and yellow) in combination. Each of these primitive colors could be assigned a weight. Weighted sums of these three primitives would be used to capture a specific color. (We know that this works from color theory.) Although relation characteristics are quite different from colors, this is essentially the same concept as is used in

RESEARCHER's relation representation scheme. It uses 5 primitives in combination to span all possible physical relations among objects. Section 4.5 describes it in more detail. Thus, a scheme based on combinations of semantic primitives is an appropriate representation system for domains that would otherwise require large numbers of simple primitives.

Characteristic descriptors can contain more than just primitives. They may have modifiers that affect an individual instance of a primitive or all primitives used in the characteristic taken as a group. In general, a characteristic representation scheme should use whatever is needed to provide sufficient capabilities to capture a relation's meaning, keeping in mind that relations are only an augmentation to a hierarchy's representation. That is, a relation characteristic scheme shouldn't be overly complex just to capture a few pathological cases.

In order to develop a relation characteristic representation scheme, the human system creator must consider the source of input data to the MERGE-based system. The system designer must be able to identify relations in the domain in which MERGE is being applied. Some sources of input data facilitate the creation of a relation representation scheme, while others hinder it. For example, corporations often publish organizational charts that show some relations among departments and executives. If these are the only relations that need to be captured, it is a relatively straight forward task to reduce them to a small set of possibilities. On the other hand, natural language input sources can have a nearly infinite number of ways of describing relations. Since the system builder can not anticipate all of them, a canonical primitive-based scheme is necessitated. (This is the case in RESEARCHER, as will be described later.)

After a reasonable number of relations in a domain have been identified, they must be scrutinized in order to develop a set of primitives. In some cases outside knowledge of what primitives to expect is helpful. The fact that most relations are binary in nature can also be an asset in characterizing the sample relations. All in all, building a complete characteristic representation scheme can be hard, but a nearly complete scheme is often adequate and attainable with a modest effort.

4.5 Other information

Relations are the primary means for augmenting an F-tree representation of a hierarchy. There are, however, other types of data that need to be present in any representational form for it to be more complete. Two basic types of information, structure-dependent and structure-independent have been distinguished.

We have defined relations to be general enough that they encompass virtually any kind of structure-dependent information. Purposes of an object's use and positional information are examples of relations according to our formalism. That is, any association among a group of objects (memettes) and some characteristic is

considered to be a relation. Structure-independent data is what will be discussed here.

Figure 4-3 shows that most of the slots in a memette are used as a means to connect it with other memettes in both the F-tree and G-trees of which it is part. The PROPERTIES slot, however, is intended to hold information specific to only the immediate memette. The structure-independent data that fills it is used to describe the memette in isolation, if the memette is unitary. If a memette is not unitary (i.e., is not a leaf node in an F-tree), the PROPERTIES slot filler applies to a collection of memettes. The collection is defined as the sub-hierarchy below the memette which has its PROPERTIES slot filled. For example, instead of having a plain vanilla disc drive, one could specify a blue titanium disc drive by filling the PROPERTIES slot accordingly. In this case, both "blue" and "titanium" apply to the color and construction of the disc drive, respectively.

Color, size, material, and other properties are encoded as structure-independent data. Note that none of these needs to reference any other object in the F-tree. Typically, they can be encoded as type-token pairs (e.g., color-***blue***, material-*titanium*). In this manner, a single slot can be used to store a variety of data types.

An ad hoc system that creates property types and tokens as needed may be adequate for some purposes. More likely, it would be useful to have a fixed set of types that are appropriate for a given domain. It may also be possible to create a system of semantic primitives to reduce the set of tokens in some cases. Our previous example of primitives to describe color could be used, perhaps. Numerical data can be grouped into ranges (as is done for UNIMEM in [Lebowitz 83c]), and material composition information might be expressed as a chemical formulation.

The variety of structure-independent data is large. Almost any technique can be used for the purpose of encoding it because it stands apart from the F-rel structure and is thus unaffected by a reorganization of the F-tree (which takes place during generalization in MERGE). Generalizations based on this type of data are the norm in most AI work. The MERGE scheme offers a way for generalizing on the structure-dependent data as well. The details of generalizing on structure-dependent data are given in the next chapter.

4.6 RESEARCHER's representation scheme

Because of the importance of non-fundamental relations to the representation of a hierarchy, we present an example of a sophisticated relation representation scheme that can serve as a paradigm for developing other such schemes.

RESEARCHER parses patent abstracts into a memette-based F-trees, among other processing. Descriptions of physical relations encountered in the text provide

information for creating and augmenting the basic F-tree. The nature of these relations are sufficiently complex that a robust primitive-based canonical scheme is needed to capture them. The relation encoding scheme used demonstrates much of what was discussed in the preceding sections. Here, we give an abbreviated account of the system developed. A more complete description can be found in [Wasserman and Lebowitz 83].

In reading over this synopsis, the reader should bear in mind that the physical objects that RESEARCHER processes are very complex. Thus, it is our goal to represent them as a reasonable approximation of reality -- we do not claim that this scheme is complete. In addition, the input is a somewhat ***unnatural*** natural language. Patent abstracts are written in legalese and hence have fewer ambiguities than conversational language, but also have a more explicit means for expressing simple ideas.

In this section, we will try to convey some of the methodology that went into developing RESEARCHER's relation representation scheme. Since each domain of application of MERGE requires a unique scheme for representing relations, the best we can hope to do is to demonstrate a paradigm for creating such schemes. By noting our methodology, another researcher should pick up some clues on how to proceed in a new domain. We begin by giving an overview of the scheme and RESEARCHER's domain. Following this, a description of the relation characteristic encoding, the heart of the relation representation scheme, is described.

4.6.1 Overview

A study of the domain of application is the first step in creating a relation representation scheme. Although our goal in developing RESEARCHER's scheme was to capture any English language description of relations among physical objects, our examples come from patent abstracts. A sample of such an abstract follows:

> **Enclosed Disc Drive having Combination Filter Assembly**
>
> **"A combination filter system for an enclosed disc drive in which a breather filter is provided in a central position in the disc drive cover and a recirculating air filter is concentrically positioned about the breather filter."**

Before showing how this text is represented in RESEARCHER, we must first describe the abbreviated memette frame that will be used as the basis for this encoding. Figure 4-5 shows the frame slots that are used in the next example. They are a subset of those shown in Figure 4-3. The MEMETTE-ID is simply the name of the physical object being described, if it is known. The TYPE slot indicates whether this is a single indivisible structure (***unitary***) or a conglomeration of pieces (***composite***). The RELATIONS slot contains a set of ***relation records***, if the memette has any relations stored in it. (In this example, relations are stored in the lowest common ancestor memette.)

MEMETTE-ID: <name-of-object>
TYPE: *unitary* or *composite*
COMPONENTS: a list of memettes if *composite*
RELATIONS: a list of <relation records>

Figure 4-5: Simplified memette frame.

The memette structure for the patent abstract shown above appears in Figure 4-6. (An equivalent representation using our compact notation is shown in Figure 4-7.) Note that some of the information encoded here is not stated explicitly in the text. For example, the *case* is TYPEd as a unitary memette; since virtually nothing was said about the enclosure, this information was assumed by the reader. Likewise, the *disc-drive* itself is considered to be composite, although this information would have had to be acquired outside the context of this example.

The RELATIONS slot stores a list of relations among individual memettes that make up the F-tree. Each relation comprises a ***relation frame*** (which is that name given to a relation's characteristic in RESEARCHER) and a list of relation arguments. As was mentioned earlier, relation characteristics (frames) can have modifiers associated with them. Here, relation modifiers appear in square brackets immediately following the frame name. At present RESEARCHER has no consistent system for describing relation modifiers. They are simply extracted from the text and mapped, via a dictionary, into a smaller set of modifiers. However, a primitive-based scheme can be of use in encoding these as well.

The major feature that becomes apparent when looking at relations in this domain, is that they are binary in nature. In English, relations are usually described by either a subject-relation-object ordering or an object-relation-subject ordering. RESEARCHER's parser determines which order is correct based on dictionary entries for each relation word. It then creates a relation record, placing the subject and object in their proper places in the representation (i.e., subject before object). There are some exceptions to this type of processing (e.g., the between relation and relations having multiple objects) but this is principally how the overall relation representation scheme works.

Our main focus, however, is on relation frames. They have been given descriptive names in this example, but there is a sophisticated primitive-based scheme behind them. Each relation word maps into a particular relation frame defined in RESEARCHER. Of course, several words that mean the same thing get mapped to the same frame, even if their subject-object ordering is different. We present a short description of this scheme next.

(MEMETTE-ID: *enclosed-disc-drive*
TYPE: composite
COMPONENTS: *disc-drive*, *enclosure*
RELATIONS: ((INSIDE-OF *disc-drive enclosure*)))

(MEMETTE-ID: *enclosure*
TYPE: composite
COMPONENTS: *cover*, *case*
RELATIONS: ((ON-TOP-OF *cover case*)))

(MEMETTE-ID: *case*
TYPE: unitary)

(MEMETTE-ID: *disc-drive*
TYPE: composite)

(MEMETTE-ID: *cover*
TYPE: composite
COMPONENTS: *breather-filter*, *cover*, *recirculating-air-filter*
RELATIONS: ((INSIDE-OF[centrally] *breather-filter cover*)
(SURROUNDED-BY[centrally] *breather-filter*
recirculating-air-filter)))

(MEMETTE-ID: *breather-filter*
TYPE: unknown)

(MEMETTE-ID: *recirculating-air-filter*
TYPE: unknown)

This shows the encoding of the partial patent abstract that appears above. Note that two of the relation frames are effected by the *centrally* modifier. Relations are stored in their lowest common ancestor memette.

Figure 4-6: Sample memette encoding.

4.6.2 Researcher's relation frames

The process of developing this relation characteristic (frame) representation scheme started by manually listing many dozens of relation description words culled from sample patent abstracts. This list was then divided and subdivided until some "natural" categorization became apparent. These categories seemed to make sense according to some intuition that we had. It is difficult to say exactly why these categories made sense. Nonetheless, the words were categorized and the search began to find some underlying physical basis that distinguished these categories. (This process is both domain dependent and subjective. We have no systematized approach to categorizing relations in an arbitrary domain.)

memettes

ENCLOSED-DISC-DRIVE:	disc-drive, enclosure
ENCLOSURE:	cover, case
CASE:	
DISC-DRIVE:	
COVER:	breather-filter, recirculating-air-filter
BREATHER-FILTER:	
RECIRCULATING-AIR-FILTER:	

relations

INSIDE-OF(disc-drive, enclosure)
ON-TOP-OF(cover, case)
INSIDE-OF[centrally](breather-filter, cover)
SURROUNDED-BY[centrally](breather-filter,
recirculating-air-filter)

This representation encodes the same information as Figure 4-6 does. Note that the relations are grouped separately, however they could have been stored in a particular memette.

Figure 4-7: Compact encoding.

One goal of a primitive-based system is that each primitive, itself, makes sense (i.e., is a physical relation in this case). We found five primitives that suffice to provide a relation frame representation for a wide range of physical relations. These five primitives are outlined in Figure 4-8. They form the basis of a combinatorial primitive scheme.

Before a description of each of these primitives and examples of their use are given, a few points about this representation scheme should be noted. First, not every relation frame has all of these five primitives. In fact most frames are adequately described by one or two of these primitives. Secondly, the fact that a relation has a particular primitive is often more important to consider than the value that this primitive takes on. In particular, the scale values for the ***contact*** and ***distance*** primitives are rather arbitrary. However, relative scale values have meaning.

A system similar to ours is presented in [Kuipers 77]. He uses a variable size frame structure to describe motion in 2-dimensional space. Some of our five primitives are analogous to the ones he uses for spatial orientation and location in his TOUR model. His work helped in deciding which primitives were needed for this scheme.

Distance is probably the simplest of the five primitives listed. It is used to indicate that two objects are separated from each other by some length. Because

primitive	description	value(s)
distance	used for relations that refer to disjoint objects. (e.g., near, remote)	a single integer from 0 to 10. 0 - close, 10 - far
contact	describes the degree to which objects are in contact with each other. (e.g., touching, affixed)	a single integer from -10 to +10. -10 = strongly forced together +10 = touching, but being forced apart
location	indicates in which direction an object is located relative to another. (e.g., above, left)	a 2D or 3D angular identification along with a reference frame indication.
orientation	describes the relative orientation of two objects. (e.g., parallel, perpendicular)	a 2D or 3D angular identification.
enclosure	used for relations which describe objects, where one is either fully or partially enclosed by another. (e.g., encircled, cornered)	*full* or *partial* plus a shape description of the interface between the enclosed and the enclosing objects.

The example words given above have been chosen to illustrate the role of each primitive and are not necessarily fully described by that one primitive alone.

Figure 4-8: Primitives of relation frames.

there seems to be an unlimited number of ways (in English) to describe distances, some method of reducing this range is needed. By forcing all distance descriptions onto a limited scale, distance relations become more manageable. In some cases, particularly in technical prose, the actual distance with some specific measurement unit (e.g., inches or meters) might be given. When this is important data, the slot values for the distance primitive can be expanded to allow for this information to be explicitly inserted.

A zero valued distance primitive would be used to indicate a relation such as "microscopically close to". On the other extreme, "astronomically far from" would certainly be a 10. A more mundane word, like "nearby", would register a 4,

perhaps. (Again, it is worth repeating that we are more interested in relative values than the actual numbers.)

The ***contact*** primitive is much like the distance primitive in that they both describe relative degrees of closeness by using a scale. It would be extremely unusual if a single binary relation required both contact and distance primitives to characterize it (i.e., they are usually mutually exclusive primitives of a characteristic).

Contact values arbitrarily range between -10 and +10. Positive values indicate that two objects are in contact but are being forced apart; the more positive the argument the stronger the force. For example, trying to pull your fingers apart, after they have been glued together, would be represented by CONTACT=+9. Negative numbers are used to indicate that the objects are in contact and are being forced together. The bond formed between two oppositely polarized magnets could possibly be valued as CONTACT=-9, while a good quality record turntable has CONTACT=-1 between the tonearm and the record (while it is being played). As with positive numbers, the larger the magnitude of the argument (i.e., more negative value), the more force is being exerted to force the objects together.

The *location* primitive is used to define relations which describe objects in everyday settings. Phrases like, "it's the building on the left, when you face the school" and "write your name on top of the paper" are good examples of the use of this primitive. In the first example, both the relative direction ("left") and the reference frame ("facing the school") are explicitly given. The second phrase has implicit in it that the student has a piece of paper with the normal orientation placed in front of him.

The appropriate values for a location primitive are a reference frame along with an angle ("left" would be 180 degrees, "top" would be 90 degrees). The frame of reference is important because a person standing at the school's front door and looking out would find the building to the right (0 degrees). Angular values, in 2-dimensions, can be any number from 0 to 360 degrees. Thus phrases like "below and to the right of" might imply an angle of 315 degrees, depending on the reference frame.

In the example, "write your name on top of the paper", it was mentioned that normal ***orientation*** of the piece of paper was implicit. The orientation primitive refers to the rotational disposition of an object about its own axis, relative to another object. What this means is that if we are talking about railroads and use the phrase, "the tracks are perpendicular to the ties", the orientation primitive of this relation would get a value of 90 degrees.

The orientation primitive is not used much in day-to-day language, but is quite useful in specifying relations in technical prose. For example, a phrase such as,

"the barrier strip running alongside the transformer" would use an orientation value of 0 degrees to express the parallelism. As with the location primitive, orientation values can be any angle between 0 degrees and 360 degrees. 3-dimensional values are also possible, but not common in natural language descriptions.

The remaining primitive, ***enclosure***, is different from the other four in that its value is a ***shape-descriptor***. A shape-descriptor is used to specify the shape of the boundary between the enclosed and the enclosing objects. For example, if "the tire encircles the wheel", then a shape-descriptor of a circle would be the appropriate value for the enclosure primitive. The reader is referred to the full description of this scheme [Wasserman and Lebowitz 83] for a discussion of shape-descriptors which are also used to describe the shape of unitary objects (memettes). Another piece of information provided is whether the enclosure is a ***full*** one or only a ***partial*** enclosure, as in the case of "a hand grasping a baseball". Although the full versus partial information can be inferred from the shape-descriptor, it is handy to have this fact readily available so that the nature of the enclosure can be easily found.

To help see how the primitives described here fit together into a relation frame, we consider several examples.

Figure 4-6 shows the use of the relation INSIDE-OF. "Inside" can take on several possible meanings, but in the context of this patent abstract the reader knows that the ***disc-drive*** is inside of the ***enclosure***. Furthermore, from our stereotypical knowledge of disc drives, we can conclude that the ***disc-drive*** is probably not in direct contact with the ***enclosure***, but is connected to it by some spacing device (which will be ignored here). Figure 4-9 shows how this relation frame is represented using just two primitives.

(REL-NAME: inside-of
ENCLOSURE: (*full* unknown)
DISTANCE: unknown

Relation characteristics are encoded in a frame format in RESEARCHER. Each primitive is given its own slot. Only the primitives that are needed to encode a particular relation frame are used.

Figure 4-9: INSIDE-OF relation frame.

(REL-NAME: on-top-of
CONTACT: unknown
LOCATION: (side-view 90 degrees)

Figure 4-10: ON-TOP-OF relation frame.

Another relation used in Figure 4-6, ON-TOP-OF, requires the use of two

different relation frame primitives. The information embodied in the ON-TOP-OF relation frame (shown in Figure 4-10) is a good example of implicit knowledge. There is no data in the sample text to help in processing what ON-TOP-OF means. In fact, the relation ON-TOP-OF was only implied by the use of a *cover*, and was not explicitly mentioned. This kind of processing is possible in RESEARCHER by having an object word (*cover*) automatically trigger the processing of a relation word (ON-TOP-OF) It seems quite natural to think of one object being on top of another when looking from a side-view. However, the frame of reference used in the LOCATION slot could have been from another perspective (e.g., lying down).

Note that in both of these relation frame representations a value of "unknown" was used. As stated before, the existence of the primitive is often of greater importance than its value. In each of these cases, since no reference to the measure of distance, the shape of the enclosure, or the degree of contact was made, "unknown" was used instead of picking an arbitrary value to quantify these primitives.

The conclusion that we have reached from using this scheme is that it generally works well. The major shortcoming of this scheme is that it has no provisions for encoding dynamic relations, only static relations can be represented. We believe that the addition of a few primitives might allow simple dynamic relations to be represented, but have not yet determined which ones are needed.

The methodology that we have adopted in developing this scheme has also worked in CORPORATE-RESEARCHER. That is, the process of listing examples, finding a categorization, and determining primitives. But CORPORATE-RESEARCHER's relations are very simple and do not constitute a good test of this approach. Nevertheless, we believe that this paradigm has application to many domains with a need for a complex relation representation scheme, and has undoubtedly been used by many AI researchers in the past.

4.7 Summary

The major issues that need consideration when choosing a representation system for hierarchies we believe are: capturing levels of detail in a hierarchy, encoding both structure-dependent and structure-independent data, and determining an appropriate formalism. MERGE uses a frame formalism in which memettes store information about each node in the F-tree as well as how it interrelates to other F-tree and G-tree nodes. The memette formalism offers the ability to distinguish among levels in a hierarchy while capturing the concept of memory chunking.

Relations are a form of structure-dependent data in that they refer to multiple memettes in the F-tree. They are defined by a characteristic based on semantic primitives. The intent of using semantic primitives is to obtain an input independent canonical encoding of relations. Structure-independent data can be captured by almost any scheme that lends itself to simple generalization.

RESEARCHER's scheme uses five primitives in combinations to achieve a natural language independent characterization of relations among physical objects. The primitives carry a scaled value in order to refine their meaning. This scheme provides a paradigm for developing a relation representation scheme for use in other MERGE-based systems.

Generalization is the process of recognizing similarities and/or differences among a set of instances and creating concepts that embody these commonalities. It is used as the basis for organizing memory in MERGE. G-trees are used to classify entire instance F-trees and all sub-trees within the instances. This reflects the fact that parts of systems are themselves systems. G-trees must continually be restructured as new instances are incrementally brought into a MERGE-based system so that they will form accurate generalizations of the data. Inheritance modified by the addition, subtraction, and substitution operations is used as a means to improve both computational efficiency and cognitive accuracy. However, due to computationally intractable problems data for one memette can only be inherited from non-conflicting G-tree sources (usually this means just one). Because most descriptions of hierarchies in the real world are not standardized, the ability to recognize that varying levels should correspond is needed. A MERGE-based system can achieve this via level-hopping.

5. Generalization Issues

5.1 Introduction

Generalization provides the framework for organizing memory so that understanding of hierarchies can take place. It is essential to the MERGE scheme Given that individual hierarchies can be represented as demonstrated in Chapter 4, generalization is the process used to join these together in an intelligent manner.

It is possible to single out generalization issues in MERGE as was done with representational issues in the previous chapter. The main issues relate to the fact that the objects being generalized about are hierarchically structured. Other issues in generalization that are independent of the representation formalism include: the availability of data to be generalized (i.e., is it spread out over time, or available all at once), and when and how data should be inherited.

Several interesting and difficult problems arise while studying how to make generalizations about hierarchically structured objects that do not occur in other generalization research (see [Michalski 83] for an overview). The main causes of these difficulties are: 1- objects are represented by a recursive method that can extend to arbitrary depth; 2- generalizations need to be created at each level in the recursive object structure, not just on the structure as a whole. Although not unique to hierarchy understanding, an important third source of difficulty is that MERGE generalizes incrementally as opposed to doing all-at-once type generalizations, and does this for a large number of instance objects.

The information that MERGE is intended to generalize about is made available in a piecewise fashion. The data for individual hierarchies are acquired in an

arbitrary sequence (i.e., not in a training sequence) and the integrated knowledge structure continually changes to reflect new information. Thus, generalization in MERGE must work incrementally. In some sense, incremental generalization is the main type of generalization that humans perform. Nature usually forces them to study one example at a time. (Of course, it could be argued that vision processing is an example of simultaneous processing of many instances.) Therefore, it is reasonable to focus on this process given that one of our goals is to try to understand hierarchies the way people do.

This chapter covers those issues in generalization that are affected by the peculiarities of hierarchies and incremental knowledge acquisition. The topics include the types of generalization that need to be carried out (e.g., on structure-dependent and structure-independent data), the implications that inherited data has on generalizing, the question of when should generalization be performed (e.g., should all possible generalizations always be made?), and other issues that will not be explored in as great a depth. We will partially describe a few algorithms which are used in RESEARCHER and CORPORATE-RESEARCHER in this chapter. In Appendix A, a detailed description of the basic F-tree matching algorithm used in these programs is given.

To put this chapter in perspective, consider that the previous one described issues and answers relating to the F-tree of a single hierarchy. This chapter describes problems that exist in constructing the G-trees that connect many instance hierarchies together. The next chapter will focus on the MERGE scheme that comes about from the interaction of the F-trees with the G-trees.

5.2 Types of generalization

Before describing the generalization issues in more detail, it is important to identify exactly the types of generalization that are used in MERGE. We distinguish three criteria: structure-dependent versus structure-independent, conjunctive versus disjunctive, and incremental versus all-at-once generalizations. The major form of generalization performed in our hierarchy understanding scheme is incremental, conjunctive, and structure-dependent.

5.2.1 Structure-dependent generalization

Structure-dependent generalizations are those that rely on structure-dependent data (e.g., F-rel links and non-fundamental relations) as their basis. Structure-independent generalizations are based on structure-independent information such as the properties of an object. The vast majority of research in generalization has

been done using what we call structure-independent data.[1] That is, instance objects are not structured such that one member of an object (assuming that it is composite) references other members -- each member is independent of the others (at least in a structural sense). MERGE creates generalizations of both structure-independent and structure-dependent data. However, structure-independent data is not an essential element of the scheme (as it is in systems like UNIMEM [Lebowitz 83c]) as it is only used on property slot values, therefore we will not explore it further. Structure-dependent information is of primary importance in MERGE -- it would not be possible to represent hierarchies without it. Therefore, generalizations based on it are of central importance in hierarchy understanding.

Hierarchies are nearly-decomposable, not completely-decomposable (as described in Chapter 4). This means that members of a hierarchy, to some degree, depend on other members -- they are structurally linked together. Therefore, it is necessary to use a generalization method that accounts for the structure of the data. Furthermore, as was mentioned in the previous chapter, parts of hierarchical systems are themselves hierarchical systems. Therefore a MERGE-based understander should capture this fact in the generalizations it creates, by using multiple G-trees.

There are two main complications in generalizing hierarchically structured data versus structure-independent data. The first is the creation of generalizations at each level in the hierarchy -- not just at the root level. The second is to maintain the integrity of the hierarchical levels even though the instances may not have exactly corresponding levels. That is, two instance hierarchies that do not match exactly as far as the number of levels each one has may still correspond closely.

The first of these problems has been described before (in Chapter 3 and 4). It simply means that generalizations are made about many levels in the hierarchies, perhaps concentrating on the basic or natural levels. For example, if two cars are found to have a generalized concept that includes an engine, body, and chassis, then a generalized concept of their common engine parts would also be created as would a concept of what comprises an ignition system (being a part of the engine). A MERGE-based system would build and maintain a G-tree for each unique object in the instances that is composite (i.e., has an F-tree structure below it). In this example, there would be one G-tree that serves to classify cars, another for engines, and yet another for ignition systems. Of course, the body, chassis, and their parts would also have G-trees. The result is that domains with complex hierarchically structured objects need many G-trees to organize their data. This complexity is largely alleviated by applying whatever basic generalization method is used in a

---

[1] This is an observation based upon a survey of the available generalization literature. Several papers have been written which classify research in generalization [Michalski 83; Angluin and Smith 82; Dietterich and Michalski 81]. None of these cite work in generalization about structured objects.

recursive manner to all levels in the hierarchy. Examples of this type of multi-G-tree creation will be given in Chapter 6.

The second complication arising from the hierarchical structure of the data to be generalized is interesting because it is not simply an extension of generalizing structure-independent data to multiple levels. The *level-hopping* problem is as follows: if two hierarchies are to be compared and one or more levels are missing from either instance, then these levels must somehow be *hopped* in the generalization process. Consider, for example, the F-trees for *corporation-a* and *corporation-b* that are shown in Figure 5-1. A generalization algorithm must recognize that the three managers, the president, and the chairman should match up in both companies. The generalized concept of a corporation would have to represent the idea that an executive vice-president may or may not be present in a stereotypical company. Of course, it may be best to assume that the executive vice-president was missing from the encoding of *corporation-b* and that it should be inherited from the generalized concept that has it. (We will take this matter up in a later section.)

CORPORATION-A	
CHAIRMAN:	president
PRESIDENT:	executive-vice-president
EXECUTIVE-VICE-PRESIDENT:	manager-1, manager-2, manager-3

CORPORATION-B	
CHAIRMAN:	president
PRESIDENT:	manager-1, manager-2, manager-3

F-trees for two similar corporations are shown. They would match exactly if *corporation-b* had an *executive-vice-president* interposed between the *president* and the managers. Level-hopping can be used to allow this match to occur.

Figure 5-1: F-trees in need of level-hopping to match.

The level-hopping problem can be a difficult one to solve. Ambiguities sometime arise as to which is the correct generalization to make when level-hopping is allowed. The diagrams shown in Figure 5-2 demonstrate that although level-hopping can lead to finding F-tree matches that would otherwise not be found, it can also create impossibly hard generalization problems. (It turns out to be combinatorially explosive.) The method for representing a possible missing level in a generalized hierarchy is important and will be described in Chapter 6. The basic algorithm for level hopping appears in Appendix A.

CORPORATION-C
PRESIDENT: executive-vice-president, general-vice-president
EXECUTIVE-VICE-PRESIDENT: manager

CORPORATION-D
PRESIDENT: executive-vice-president, manager
EXECUTIVE-VICE-PRESIDENT: general-vice-president

CORPORATION-#1
PRESIDENT: executive-vice-president, ?-1
?-1: ?-2

CORPORATION-#2
PRESIDENT: ?-3, ?-4
?-3: manager
?-4: general-vice-president

The top two diagrams show corporate F-trees that are somewhat similar. If they are generalized without the use of level-hopping the result is the F-tree shown as *corporation-#1*. Memettes ?-1 and ?-2 represent corporate officers that have been matched in *corporation-c* and *corporation-d* but are not of the same type. If level-hopping is allowed, the resultant generalized corporation (*corporation-#2*) has a different structure than *corporation-#1*. The ?-3 and ?-4 memettes represent corporate officers that may or may not really exist. Note that level-hopping allows more data to be matched exactly, but it is not clear that it gives the best result.

Figure 5-2: Generalization with and without level-hopping.

5.2.2 Conjunctive generalization

The type of generalizations that a MERGE-based system makes are of the conjunctive variety. All of the variants of a generalized concept must include the data that the generalized concept contains (modulo the subtraction and substitution operations). A generalized concept is the logical intersection of its variant concepts and instances. This imposes certain constraints on the way information is represented in the G-trees (e.g., there is no way to represent an alternate choice). Therefore, it is important to understand what is possible using conjunctive generalizations versus other methods.

The obvious alternative to conjunctive generalization is disjunctive generalization. We have defined disjunctive generalization to be the formation of a concept that includes all the data in the union of its variant concepts and instances. (Actually, "disjunctive generalization" is somewhat of a misnomer because a disjunction (union) does not find commonalities or differences within a set of data, while the word "generalization" implies exactly this.) This method, by itself, would also impose an

equivalent number of constraints on the G-tree representations. By using both conjunctive and disjunctive generalizations simultaneously these constraints would be lifted.

It seems self-evident that people make both conjunctive and disjunctive generalizations. For example, a typical generalization might be: "a disc drive has a motor, a read/write head, and either a floppy disc or a hard disc". But now suppose that a new data storage media is invented, called a ***semi-hard disc***. Should the generalization become: "a disc drive has a motor, a read/write head, and either a floppy, a hard, or a semi-hard disc", or should it simply be: "a disc drive has a motor, a read/write head, and some kind of disc"? (It is interesting to note that "some kind of disc" requires the ability to generalize about pieces of an object. MERGE allows this to be done because of its recursive method of generalization down through the levels in an F-tree.) The answer is not clear; these two alternatives appear equally valid in this context. However, as the number of different instances of disc media grows, the generalization should probably become the simpler one. But how many different possible instances must be known first?

The problem of which generalization to make arises because of the ***lack*** of constraints provided by combining conjunctive and disjunctive generalization. If only conjunctive generalization were permitted, for example, there would be no choice of generalization to make. Humans appear to be able to handle this choice and probably pick the "best" generalization for a given application. We do not use both conjunctive and disjunctive generalizations in MERGE mainly because of the increased complexity of having to choose among alternate correct generalizations. (There is also some motivation for constraining the types of generalization permitted based upon principles from hierarchy theory [Sussman and Steele 80; Pattee 73; Simon 73]. Removing these constraints can cause hierarchical growth to be retarded unless there is some other, higher level, organizing structure.)

Allowing only conjunctive generalizations to be created can be too constraining if not modified slightly. If a large number of instances have a particular part in common and only one or two instances don't have it, then the generalized concept could not include this part. By permitting deleted-inheritance (i.e., the subtraction operation) this part can be included in the generalization and can be deleted from the counter example instances. Similarly, the substitution operation can help in allowing conjunctive generalizations to capture a wider class of concepts.

5.2.3 Incremental generalization

Incremental generalization is the mode in which MERGE functions. It is instructive to investigate the nature of this type of generalization versus the all-at-once approach because incremental generalization is a major source of difficulty and complexity in creating intelligent information systems.

In the all-at-once approach, all instance F-trees are compared simultaneously and a resultant hierarchy classifying them is created. Generalizing by incremental amounts is a second possibility. Incremental generalization is the process by which each new instance F-tree is compared against the current G-trees. Each F-tree node then becomes a variant of some pre-existing G-tree node or it may be used to produce a new generalization under which it is stored.

The distinction between all-at-once type generalization and incremental generalization is not specific to the problem of hierarchy understanding. The choice of method of generalization used in any situation is usually heavily dependent on the domain being investigated. The most obvious reason for choosing one or the other method depends on whether the instance data is available all at one time or is supplied piecemeal to the system. Obviously, if a state description (generalization hierarchy) of the currently known data is needed on a continual basis and the data is trickling into the system over a period of time, then incremental generalization must be used. On the other hand, if a system's task is to classify a given set of instances such as the various types of personal computers on the market at this moment, then an all-at-once type generalization process might be best [Michalski and Stepp 83a].

Most hierarchical phenomena that humans perceive daily are realized piece-by-piece. Over a period of time people are able to develop pictures of what a hypothetical hierarchy might look like from analyzing instances of these hierarchies. For example, a person might be overwhelmed by trying to figure out the information presented by a large corporation's organizational chart, if this were his first experience in understanding corporate structures. But after seeing a few of these (or having heard about a few different companies) he would realize that most of them have a very similar structure. This is possible because he creates a picture of a generalized corporate hierarchy. Thus, it seems natural to use incremental generalization for understanding hierarchies in real-world domains.

Virtually all of the examples of hierarchies mentioned in the this paper are open-ended domains. That is, they are continually being expanded by new instances that are not yet known. Furthermore, the real world (and our own mental limitations) usually allow us to view only one example of a hierarchy at a time. It is for these reasons that we focus our attention on incremental generalization rather than the all-at-once generalization process. There are several problems that surface by using incremental generalization as opposed to the all-at-once approach.

The most obvious of these problems is that a system may have to reorganize its stored knowledge each time a new instance is encountered. To see how this is possible, assume that we have a memory structure as shown in Figure 5-3.

Next, a new description of a corporate president is introduced. *President-c* has control of a sales department and an acquisitions department. Assuming that our

PRESIDENT-#1: finance, sales, manufacturing,
>president-a, >president-b
PRESIDENT-A: +legal
PRESIDENT-B: +planning

The concept *president-#1* has been created by generalizing *president-a* and *president-b*. (All of the F-children in both the *president-a* and *president-b* F-trees are corporate departments.)

Figure 5-3: Incremental generalization (initial memory).

goal is to make the maximal number of generalizations (i.e., recognize all commonalities), using only one VARIANT-OF link per memette, then a new concept must be created.

PRESIDENT-#2: sales, >president-#1, >president-c
PRESIDENT-#1: +finance, +manufacturing, >president-a,
>president-b
PRESIDENT-A: +legal
PRESIDENT-B: +planning
PRESIDENT-C: +acquisitions

A new concept, *president-#2*, has been created to capture the fact that all three instances have a *sales* department. Because *president-#1* is a VARIANT-OF *president-#2* it inherits the *sales* department and thus its structure must be changed to reflect this. This change could not have been anticipated until the existence of *president-c* was made known.

Figure 5-4: Incremental generalization (after memory reorganization).

Figure 5-4 shows how memory appears after the incorporation of *president-c*. Note that the concept of *president-#1* has been modified to become a VARIANT-OF *president-#2* thus adding an another level to the G-tree. If all three instances of presidents had been known to the generalizer simultaneously there would have been no need to restructure memory -- it would have been built correctly the first time. All-at-once type generalizations are routinely done in numerical taxonomy (see [Ben-Bassat and Zaidenberg 84] for an interesting example using semi-structured objects). A MERGE-based system must potentially go through the process of memory reorganization each time a new instance is incorporated.

Another problem caused by the use of incremental generalization is the possibility for making erroneous generalizations. It is not hard to imagine how a system (or person) can be misled into making incorrect generalizations because of only having a limited selection of instances available for analysis. The result of this is often a system that has built a bad classification hierarchy; one that has unreasonable concepts in it so that it will mis-classify all subsequent incoming data. This is an

area which has particular importance to a MERGE-based system that is fed instances in an arbitrary order. We do not address this problem directly because it is somewhat peripheral to this thesis and requires a great deal of study (i.e., a thesis within itself). However, the reader is referred to work done by other researchers, most notably in [Lebowitz 82].

The final problem stemming from the use of incremental generalization that we will mention is intimately tied to another issue in generalization. Incremental generalization contributes to the complexity of inheritance, which is the subject of the next section.

5.3 Generalization and inheritance

Once a generalization is created, the question of how to store this learned knowledge presents itself. Ideally, a representation scheme should allow this knowledge to be encoded so that it need not be regenerated at a later time. Having to perform the generalization process each time this piece of knowledge is required would be too time consuming in an intelligent system. Inheritance is a way of avoiding having to reproduce a generalization by arranging memory so that the results of a generalization are apparent. It also has the additional benefit of being memory efficient. Inheritance is certainly not a new idea, it has been used by countlessly many AI researchers both past and present (see [Winston 72; Fahlman 79, Brachman 79a] for example). Nevertheless, it has major significance in integrating representation and generalization.

Without some sort of inheritance scheme, a G-tree would serve only as a device to index F-trees. Each F-tree, embedded in the G-trees that comprise the knowledge base, would have to contain all the information needed to fully describe the concept or instance hierarchy. That is, an F-tree would be represented exactly the same when it is incorporated into the G-trees as it would in isolation. In addition, in order to determine what is unique about a particular F-tree, it would have to be compared to all its ancestors in the G-trees.

Given that we need to implement some form of inheritance scheme, the following questions surface: what information should be generalized and inherited, where should it be inherited from (i.e., should there be multiple sources for inheritance), and are there any special requirements for an inheritance scheme for hierarchies? We address these three questions in order.

5.3.1 Inherited information

Obviously there can be no inheritance of information that has not been captured by a generalization first. There must be some generalized concept created so that information can be inherited from it. Therefore, we will examine how inheritance is performed for each of the three types of generalized concepts that can be created in MERGE. These are the same as described in Section 5.2.

In MERGE, F-rel links are the most important form of information to generalize and inherit via the VARIANT-OF links of the G-tree. F-children at all levels in the F-tree are inherited. For example, if a concept of a disc drive contains a power supply, which is a complex object requiring a multi-level F-tree to represent, then all its variants would inherit the entire sub-structure that represents the power supply. It is possible (even likely) that all variants of a disc drive would not have exactly the same power supply. In such cases, it is desirable to have some method for permitting minor modifications to the inherited parts. The addition, subtraction, and substitution operations serve this function. By using these operators, the range of possible generalizations is increased because instances can become variants of generalized concepts that are not exactly the logical conjunction of the data stored in their variants. For example, a generalized concept of a disc drive may have a particular power supply included in it because most of its variants (instances) have that same power supply. However, the substitution operator can be used to allow one or two instances to have different power supplies yet still remain variants of the same generalized concept. This is discussed in more detail later in this section.

Relation information is the next most crucial in the representation of hierarchies. Inheritance of relations is somewhat more complicated than inheritance of F-children. Both the arguments and the characteristic of the relations can be generalized and, therefore, inherited. Generalizing and inheriting relation arguments is straightforward and similar to F-children inheritance. Characteristics can be generalized either by a simple matching of their names or by comparing their primitive representations. If a primitive comparison is used, then generalized relation characteristics can take on meanings that are not exactly the same as their variant characteristics. This may be desirable in some complex relation representation systems (as is the case in RESEARCHER).

To see how relations can be generalized, consider the following sample relations as shown in Figure 5-5. If we have the RESEARCHER relations ABOVE(disc, motor) and ON-TOP-OF(disc, spindle) they would generalize to be R(disc, x). In this case x is a place holder variable and R represents a relation characteristic with the primitive common to the ABOVE and ON-TOP-OF characteristics, namely the ***location 90 degrees*** primitive. The ABOVE relation could inherit the ***location*** primitive from the R relation as well as inheriting the *disc* argument. It would substitute the ***motor*** argument for the x argument and add the ***distance*** primitive to complete the characteristic description. The combination of inheritance, substitution, and addition results in the correct encoding of the ABOVE relation. (Note that the ABOVE characteristic requires both the ***location*** and ***distance*** primitives to be represented.) The ON-TOP-OF relation would be encoded in a similar manner using inheritance modified by substitution and addition.

In this example, it is not clear that using inheritance buys anything in terms of efficiency. But one could imagine that relations in other domains contain more information. In such cases, inheritance of relations would be space efficient.

(REL-NAME: above
LOCATION: (side-view 90 degrees)
DISTANCE: unknown

(REL-NAME: on-top-of
CONTACT: unknown
LOCATION: (side-view 90 degrees)

Two relation frames are shown. They have the LOCATION primitive in common, therefore a generalization of them would also have this primitive.

Figure 5-5: Relation frames.

However, the usefulness of inheritance as a means of determining similarities and/or differences remains, regardless of the amount of information in the relations themselves.

The remaining basic information type in MERGE is structure-independent data. In this case, generalization and inheritance is very easy, although not particularly interesting. Properties of an object such as color, melting point, and size are simple to generalize if only an exact match of variants is permitted. (This is largely what other AI systems have done (see [Winston 72; Lebowitz 83c; Michalski and Stepp 83a], for example.) It is slightly more difficult to generalize if approximate matching is used by means of primitive classes (color might have the classes red, blue, and yellow, for example) or ranges of data (for melting point temperature). Inheritance of this generalized data can proceed in the same way as relation characteristics are handled. Modifications via the substitution operator can specify a color shade or narrow a data range to a specific value.

5.3.2 Multi-source inheritance

A basic rule of inheritance in MERGE is that memettes can inherit data from multiple parents (i.e., can have multiple VARIANT-OF links) only if the data being inherited has no chance of conflicting with each other. Without this restriction, it is possible that ambiguities can be created that would require a major reorganization of the memory structure to circumvent. Just determining if a reorganization is needed turns out to be an intractable problem if a large knowledge

base is used. We call this the *multi-source* inheritance problem.[1]

To look at the multi-source inheritance problem we will use an example from the corporate world adapted from [Wasserman 84]. Four very simple (two level) hierarchies are used. Figure 5-6 describes the structure of these four companies' vice-presidents.

job title	symbolic F-tree
v.p.-company-1	VP-1: dp, ds, dm
v.p.-company-2	VP-2: da, ds, dm
v.p.-company-3	VP-3: dp, da, ds, dm
v.p.-company-4	VP-4: dp, da, dm
director-public-relations	DP:
director-advertising	DA:
director-sales	DS:
director-market-research	DM:

Four F-trees are represented here, one for each vice-president that will be used in our examples. Short mnemonics are assigned to each job title to condense the representations. Nodes *vp-1*, *vp-2*, *vp-3* and *vp-4* comprise the instances for the example followed in the next 4 figures.

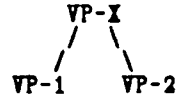
Figure 5-6: Summarized data on four companies.

We start with the generalization structure built by comparing *vp-1* with *vp-2*. Next, *vp-3* will be added into the generalization hierarchy. Finally, the full magnitude of the problem will surface when *vp-4* is added.

Figure 5-7 shows how both *vp-1* and *vp-2* can be represented as variants of a generalized object (*vp-x*) which has the F-children *ds* and *dm*. A new G-tree node has been created that captures the data that its two variant nodes have in common. It is helpful to think of the representation shown as *factoring out* what *vp-1* and *vp-2* have in common and storing it in node *vp-x*. With this in mind consider what happens when company-3's F-tree is added into the G-tree.

The new G-tree node, *vp-3*, shown in Figure 5-8 demonstrates how its data can be encoded by using inheritance from two sources. (These two sources are actually

[1]The problems associated with inheritance from multiple sources have been addressed to some extent before [Brachman 79b]. The solution presented by Brachman is to use special *modification links* to connect nodes that might cause conflicting data to be inherited from multiple sources. We have a similar solution via the subtraction and substitution operators. However, our focus is on the more theoretical problem of how to inherit from multiple sources without resorting to modifications. In particular, we want to know if multi-source inheritance can be permitted when incrementally generalizing about a large number of hierarchically structured objects.

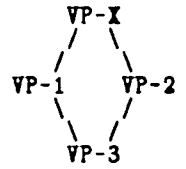



```
VP-X: ds, dm, >vp-1, >vp-2
VP-1: +dp
VP-2: +da
```

Company-1's vice-president (*vp-1*) and company-2's vice-president (*vp-2*) have been generalized, creating a new G-tree node (*vp-x*) that contains the information that *vp-1* and *vp-2* have in common. (The left side of this diagram, and the next three that follow, shows the basic G-tree structure in the unified memory structure. We provide this as an aid in helping the reader understand the multi-source inheritance problem.)

Figure 5-7: Generalization of company-1 and company-2.

part of the same global G-tree but can be equivalently thought of as independent G-trees.) There is also no ambiguity as to what node *vp-3* should inherit. Obviously it should only have one F-rel link to *ds* and one F-rel link to *dm*. Since it inherits these links from the same *ultimate* source (namely *vp-x*), there is no problem in determining if it should have two copies of *ds* and *dm* or one copy. This can be done by following the VARIANT-OF links in the unified memory structure until an intersection in the G-trees is found. Unfortunately, this this is not always straightforward.



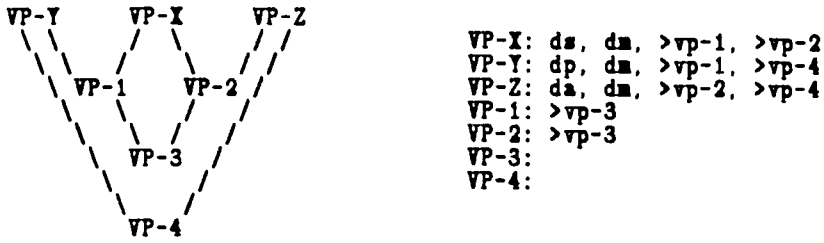

```
VP-X: ds, dm, >vp-1, >vp-2
VP-1: +dp, >vp-3
VP-2: +da, >vp-3
VP-3:
```

This G-tree representation now includes company-3's vice-president. Since node *vp-3* is just the union of the data in nodes *vp-1* and *vp-2* no new F-children have been added to *vp-3*'s F-tree. This is an example of inheritance from two sources that causes no problems.

Figure 5-8: Addition of company-3 into the G-tree.

In Figure 5-9 we have attempted to incorporate *vp-4* into the G-tree. Two new G-tree nodes have been created. *Vp-y* and *vp-z* represent objects that have F-children *dp*, *dm*, and *da*, *dm*, respectively. Node *vp-4* is a VARIANT-OF both *vp-y* and *vp-z* in that it has F-children *dp*, *da*, and *dm*. Aside from having the problem of determining if *vp-4* should have *dm* as an F-child once or twice, this memory reorganization has also created problems with the representations of the other three vice-presidents. The problem that arises here is what F-children do nodes *vp-1*, *vp-2*, *vp-3*, and *vp-4* really contain?

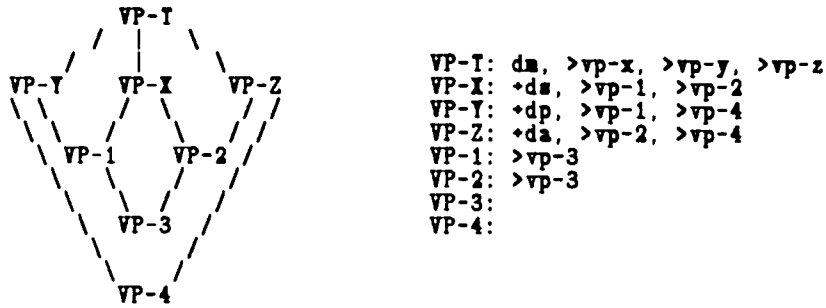
The solution is found by keeping in mind that we would like to factor out the common elements in G-tree nodes. To accomplish this we factor out the *dm* node from *vp-y* and *vp-z* then node *vp-4* will not have any ambiguities (i.e., *vp-4* will inherit only one F-rel link to node *dm*). Recognizing that node *vp-x* contains *dm* as a factor, we also take it out by making *vp-x* a VARIANT-OF a new node, *vp-t*, which contains only *dm* as an F-child. Next we make *vp-y* and *vp-z* variants of



A first attempt at incorporating *vp-4* into the G-tree shown in Figure 5-8. It fails because of multi-source inheritance ambiguities. For example, it is unclear if *vp-4* has one or two F-rel links to *dm*.

Figure 5-9: A first try at incorporating company-4 into the G-tree.

vp-t, as well. This will result in a structure that has only a single node that contains *dm*, wherein all other nodes that need this as an F-child will inherit *dm* from this ultimate source. Figure 5-10 demonstrates exactly this.



Finally all the ambiguities have been eliminated. By factoring out *dm* from nodes *vp-x*, *vp-y*, and *vp-z*, node *vp-t* has become the *only* source of *dm* for all of the nodes (vice-presidents) that have *dm* in their F-tree (employ). The key concept is: all common factors must be singled out to form multi-source inheritance hierarchies that are ambiguity-free.

Figure 5-10: Final G-tree representation of all four companies.

Left to explain is how we determined that *dm* had to be factored out from *vp-x* to fix the problem? Furthermore, how did we even know that we had to check nodes *vp-1*, *vp-2*, and *vp-3* for potential problems? The answer is we didn't. With a little bit of thought one can reach the conclusion that each time a new generalization is made it is possible that this factoring problem might cause some previously represented node to become mis-represented (or at least ambiguous in meaning). To state this more precisely: if a new generalization is built that breaks up some previously existing group of factors (i.e., the F-children in a generalized concept), then it is possible that one or more nodes in the representation will inherit the same F-children from more than one ultimate source.

The consequence of this finding is that MERGE-based systems cannot allow for inheritance from multiple sources where the potential for ambiguity exists. From a human cognition perspective, it may seem acceptable (perhaps even necessary) to

permit this kind of inheritance. However, it becomes computationally intractable in any reasonably large knowledge base. There are various compromises that can be made to allow for certain aspects of multi-source inheritance to be performed while still maintaining rapid processing speed. They are discussed in [Wasserman 84]. RESEARCHER and CORPORATE-RESEARCHER simply take the approach that multi-source inheritance is not allowed in any form.

5.3.3 Hierarchical inheritance

We have already touched on the significance of inheritance to an intelligent information system. Our particular type of intelligent information system (MERGE) is designed to understand hierarchies. It is therefore useful to identify what aspects of inheritance are important when applied to generalizing hierarchies.

A prominent feature of a hierarchical domain with many instances is that its individual hierarchies usually have much in common. This is particularly true of man-made hierarchies. Complex physical objects such as automobiles, disc drives, and stereo receivers have the property that most of the instances of each item have the same components regardless of the manufacturer or model of the item. An examination of a couple of floppy disc drives would show that many of the subassemblies are exactly the same. Other man-made hierarchical systems have this property. Corporations often borrow organizational arrangements from other companies, for example.

The reason why man-made hierarchies tend to have many common sub-hierarchies seems obvious. The basic philosophy involved is the mathematician's paradigm of "reducing a problem to one that has already been solved". An engineer designing a disc drive might be faced with the problem of how to position the read/write head on the disc. Using this paradigm he would immediately solve his problem if he had previously designed such a mechanism. If not, he could break his task into smaller sub-problems and see if any of these had been solved before, otherwise he could look for someone else's solution.

The key ingredient in creating an inheritance scheme for hierarchy generalization, therefore, is to allow for common subassemblies to be stored in generalized concepts and to permit minor modifications to them. We have described how the substitution operation can accomplish this but we have not looked at it in detail. The following example should make this process clear.

Assume that we have two F-trees each describing a floppy disc drive as shown in Figure 5-11. (In this example we explicitly identify each memette according to what F-tree it is initially a part of (e.g., *motor-2* is in *floppy-disc-drive-2*. This allows us to carefully follow the results of the generalization process.) Their generalization is a floppy disc drive that is conceptually identical to *floppy-disc-drive-1*. The only difference between the two is that *floppy-disc-drive-2*

has an extra coil in its motor. Figure 5-12 shows the generalized concept, *floppy-disc-drive-#*, with its variants as a unified memory structure.

FLOPPY-DISC-DRIVE-1: drive-assembly-1, r/w-assembly-1
R/W-ASSEMBLY-1: carriage-1, r/w-head-1
DRIVE-ASSEMBLY-1: motor-1, spindle-1
MOTOR-1: coil-1

FLOPPY-DISC-DRIVE-2: drive-assembly-2, r/w-assembly-2
R/W-ASSEMBLY-2: carriage-2, r/w-head-2
DRIVE-ASSEMBLY-2: motor-2, spindle-2
MOTOR-2: coil-2a, coil-2b

The F-trees of two floppy disc drives are shown. The only difference between them is that *floppy-disc-drive-2* has an extra coil in its motor.

Figure 5-11: Two similar floppy disc drives.

FLOPPY-DISC-DRIVE-#: drive-assembly-#, r/w-assembly-#,
>floppy-disc-drive-1, >floppy-disc-drive-2
R/W-ASSEMBLY-#: carriage-#, r/w-head-#
DRIVE-ASSEMBLY-#: motor-#, spindle-#, >drive-assembly-2
MOTOR-#: coil-#, >motor-2
FLOPPY-DISC-DRIVE-1:
FLOPPY-DISC-DRIVE-2: drive-assembly-#-+drive-assembly-2
DRIVE-ASSEMBLY-2: motor-#-+motor-2
MOTOR-2: +coil-2b

Floppy-disc-drive-# is the generalization of *floppy-disc-drive-1* and *floppy-disc-drive-2*. The differences that *floppy-disc-drive-2* has from its parent are encoded by use of the addition operator (at the lowest level) and the substitution operator (at higher levels).

Figure 5-12: The generalized floppy disc drive.

This knowledge structure has captured the fact that both instances have an identical read/write assembly and so inherit *r/w-assembly-#* intact. The extra coil (*coil-2b*) in *floppy-disc-drive-2* is shown as an additional F-child of *motor-2*. As a consequence of this, *drive-assembly-2* must differ somewhat from *drive-assembly-#*. It is made a variant of the generalized drive-assembly (*drive-assembly-#*) with the substitution of *motor-2* for *motor-#*. Note that, although *drive-assembly-2* modifies *motor-#* it inherits *spindle-#* as is. In addition, *floppy-disc-drive-2* must use the substitution operation to replace *drive-assembly-#* with *drive-assembly-2*.

The overall effect of inheritance with the substitution operator in a hierarchical domain is to represent the idea that: If an object differs from a standard at a low level of detail then it must also be different at all levels above that (but only the differences at each level are recorded).

5.4 When and where to generalize

We have described our method for representing generalizations of hierarchies. Although a formalism for representation and generalization has been presented, we have yet to determine how often to create generalizations. The matter of where a new instance hierarchy should be incorporated into a G-tree also has to be addressed. In this section, we discuss several issues relating to when and where to generalize within the G-tree.

The MERGE scheme is primarily intended as a large scale organizational mechanism for memory. Our main focus in developing MERGE is in the interplay between representations of single hierarchies and generalizations based upon them. The concepts and heuristics needed to achieve this overall knowledge structure are certainly important. However, the details of an algorithm for a specific implementation of MERGE are not of central importance, here. The reader should consider what follows to be an overview of concepts dealing with when to create new generalizations and where to place them in the G-tree.

The issues presented here are: how incomplete information in instances should be processed, when to make new generalizations, and how to locate the best place in memory to store generalizations.

5.4.1 Incomplete information

In the real world, most descriptions of complex objects are incomplete. Sources of data that describe instances of hierarchies do not usually contain sufficient information to detail all members of a structure. (This is particularly true for natural language input sources, as is the case in RESEARCHER.) A single description usually concentrates on specifying either a few levels in a hierarchy or a particular sub-hierarchy. For example, corporate charts often show only the upper level management organization, and disc drive patents usually only give a detailed description of subassemblies that make the device patentable. For this reason, it is necessary to design an understanding system to be robust and intelligent in order to account for missing information.

The basic dilemma faced is whether to assume that non-specified information actually exists (and was not included in the description) or simply does not exist. If a particular instance is incompletely described and the missing data is contained in a previously created generalized concept then it can be inherited from that concept. Of course, it is possible to create an erroneous representation if such *default* inheritance is done.

In some cases, the inheritance of a default value for missing information seems correct. For example, if a natural language description of a television set failed to mention a picture tube, it would almost certainly be correct to assume that one was there. The reason this assumption is valid is because we have seen and read about many instances of televisions and have built the generalization that all televisions have a picture tube. Furthermore, every instance of a television set we have encountered has had a picture tube (except for some very recent LCD sets) making this a *strong* generalization -- one that has an impressive number of instances that support it.

A *weak* generalization has fewer variants backing it, or it is supported by instances with incomplete information. An example of one such generalization might be: "large corporations have an acquisitions department". One may know of some corporations with acquisitions departments, and have few, if any, known counter examples (i.e., know that a company explicitly does not have an acquisitions department). It is likely that the above generalization would have been made based on instances where the presence of an acquisitions department has been assumed. Inheriting defaults from weak generalizations is not necessarily a good idea as it can produce wrong results (see [Abelson 73; Carbonell 81] for a further discussion of the use of weak generalizations).

With no other information about a domain, the heuristic for deciding whether or not to inherit missing data should be based on the strength of the parent generalization. If it is a strong generalization with a large number of variants (according to some cognitive criteria) then default inheritance should be allowed. This would work well with G-trees that have broad spans because there would be the possibility of having a high degree of confidence in the generalization (see work on UNIMEM [Lebowitz 83c] for a discussion of confidence). Unfortunately, this approach is problematic for narrower span G-trees, especially when few instances are present in the G-tree, as is the case when starting up a new system. (The problem of instance example ordering, when starting up a new system, is discussed later in this section.) If, for example, there is only one other known instance of a hierarchy then there is insufficient reason to believe that a new instance should conform to it by allowing default inheritance. In fact, it should mean that the missing part is equally likely to be present or not in that hierarchy. In this particular case, since MERGE can not directly represent disjunctions, it would use the subtraction operation to represent missing information.

5.4.2 When to make generalizations

In Section 5.2, we discussed the types of data that MERGE generalizes and how inheritance is used for each type. Throughout that section, and in others, we have skirted the question of how much generalization should be done. Do we want to make all possible generalizations or only some, and which ones should be made?

We have, in some sense, already answered this question. Since MERGE allows memettes to be variants in only one G-tree per F-rel type, all possible generalizations are not made (i.e., we do not allow multi-source inheritance). However, it is permissible to have memettes classified in multiple G-trees based on different F-rels (a G-tree based on a FUNCTIONAL F-rel, for example). In this case, the question of which generalizations to make is still open.

There is also the question of whether to create a new generalization if a group of F-trees have very few differences among them. This too is an open question, but its answer is mostly affected by the particular application for MERGE.

A good principle to use in order to find solutions to these questions is to examine a few examples of how humans generalize. When people see a new car model it appears that they make few, if any, new generalizations unless there is something unique about the car. The first time someone sees a hologram that person probably has a difficult time making any sense out of it because it is unlike most other objects that he has encountered. Personal computer systems are still in a period of rapid change and people have to update their generalizations about them each time they see or read about a new one (unless, for example, it's claimed to be IBM-PC compatible). These examples demonstrate that a rule for how much to generalize seems to be something like: "If I know a large amount about a class of objects (or know very little about them) then the less I need to (or can) generalize about a new instance in this class".

MERGE implicitly embodies this principle. The first instance brought into a MERGE-based scheme cannot be generalized against anything. As more instances are incorporated into the knowledge structure, there are increasingly more possible generalizations that can be made. At some point, however, the number of possible generalizations starts to decrease. The reason for this is that the likelihood of the needed generalizations having already been made increases with each new instance. (This assumes that the state of the world knowledge that a system is trying to perceive is static.) Since complex objects tend to have common sub-hierarchies, the chances of finding a specific sub-hierarchy is greater as the knowledge base grows. The problem of how much to generalize, then, is shifted to one of locating where in the G-tree the needed generalizations have already been created. This, too, is a difficult problem, but we give one possible solution below.

5.4.3 Locating generalizations

In a MERGE-based system, many G-trees exist; each one serving to classify a different object (i.e., sub-hierarchy). Usually the system will be used to understand the root object that the input F-trees describe. In other words, the top-level memette in an F-tree is the most important one. When generalizing, it is this memette that is used as the key in locating the proper place to store a generalization. Thus, the generalization location process need only look at the G-

tree that classifies this key memette. In the following description, it is this G-tree that is being referred to.

The G-tree is a structure that is built up incrementally as each new F-tree is brought into the system. As such, it dynamically changes over time. The concepts it embodies are continually changed to reflect the current knowledge state. Each new instance must be indexed somewhere in this structure. There are potentially many locations where a new F-tree can be stored in the G-tree. Finding the "best" place is dependent on the criteria for deciding how good a generalization is. A metric for determining how good a generalization is described in the next section when F-tree matching is discussed.

Although we will not discuss optimal algorithms for finding the "best" place to store an instance hierarchy, we will need some algorithm to demonstrate how a G-tree is incrementally created. A simple, but useful, algorithm is to start at the root of the G-tree and follow the variant link to the F-tree that gives the "best" generalization (locally). This continues, recursively, until no better generalization is found. At this point, a new concept is built, if necessary, using the addition, subtraction, and substitution operators. (This is the algorithm that RESEARCHER and CORPORATE-RESEARCHER currently use.)

Consider the F-tree representations shown in Figure 5-13. (We have used F-trees similar to those used to demonstrate the multi-source inheritance problem, although these issues are unrelated.) We begin this demonstration of the generalization location process by initializing the knowledge base with the generalization of *vp-1* and *vp-2*, as shown in Figure 5-14(a). The generalized concept, *vp-x*, has been created, and both *vp-1* and *vp-2* have been indexed as variants of it.

When *vp-3* is incorporated into the G-tree, the location determining algorithm finds that it most closely corresponds with *vp-2* and so creates a new concept, *vp-y*, that captures the common elements of *vp-2* and *vp-3*, as shown in Figure 5-14(b). The algorithm was able to find this by first comparing *vp-3* against the G-tree root, *vp-x*. It then tried to match *vp-3* against each variant of *vp-x*. Finding that *vp-2* gave the better match, it then was able to build *vp-y* into the G-tree at the correct location. The F-tree for *vp-4* can be incorporated into the G-tree without any added generalizations; concept *vp-y* is already the most specific generalization for it in the context of *vp-1*, *vp-2*, and *vp-3*. Figure 5-14(c) shows that F-tree *vp-4* is simply indexed as a variant of *vp-y*.

This demonstrates the point that fewer generalizations are needed as the knowledge base grows and that finding the correct place in the G-tree is equivalent to determining how much generalization needs to be done. Specifically, once the correct place is found, all the work for generalization has already been done.

In the real world, one cannot usually pick and choose the order in which instances

VP-1: dp, ds, dm
VP-2: ds, dm, da
VP-3: dm, da, df
VP-4: dm, da, dl

Four simple F-trees are shown. Each contains a set of three directors that will be used in the next two examples. These are similar to the F-trees used in Figures 5-6 through 5-10, but comprise an unrelated example. A director-of-finance (*df*) and a director-of-legal-services (*dl*) have also been added.

Figure 5-13: Data for incremental location examples.

VP-X: ds, dm, >vp-1, >vp-2
VP-1: +dp
VP-2: +da
(a)

VP-X: ds, dm, >vp-1, >vp-y
VP-Y: ds–+da, >vp-2, >vp-3
VP-1: +dp
VP-2: +ds
VP-3: +df
(b)

VP-X: ds, dm, >vp-1, >vp-y
VP-Y: ds–+da, >vp-2, >vp-3, >vp-4
VP-1: +dp
VP-2: +ds
VP-3: +df
VP-4: +dl
(c)

F-trees *vp-1* and *vp-2* have been generalized creating concept *vp-x* (shown in diagram (a)). Instance *vp-3* is found to be most similar to *vp-2* thus necessitating a new generalization, *vp-y*, as shown in (b). F-tree *vp-4* can be stored as a variant of *vp-y* without any new concepts, as depicted in (c). In the process of locating where *vp-4* should be stored it was determined that no new generalizations would be needed.

Figure 5-14: Locating and storing new F-trees in the G-tree.

become available to an understanding system. It is, therefore, desirable that a system be able to give satisfactory results regardless of the instance ordering it is presented with.

We can use this example to demonstrate the effect of instance ordering when incremental generalization is performed. Instead of feeding the generalization

algorithm the F-trees in the order *vp-1*, *vp-2*, *vp-3*, *vp-4*, we use the order *vp-2*, *vp-3*, *vp-4*, *vp-1*. The results of doing so are shown in Figure 5-15. Note that the final G-tree in this figure (Figure 5-15(c)), rooted at *vp-p*, and the one with root *vp-x* (Figure 5-14(c)) are equally valid, but organize the same information differently. They differ because they have been "forced" to create concepts incrementally instead of waiting for all instance F-trees to be available. The concepts created early on will affect the way a system learns subsequent information.

VP-P: dm, da, >vp-2, >vp-3
VP-2: +ds
VP-3: +df

(a)

VP-P: dm, da, >vp-2, >vp-3, >vp-4
VP-2: +ds
VP-3: +df
VP-4: +dl

(b)

VP-P: dm, da, >vp-3, >vp-4, >vp-q
VP-Q: da–+ds, >vp-1, >vp-2
VP-1: +dp
VP-2: +da
VP-3: +df
VP-4: +dl

(c)

The same F-trees are used here as in Figure 5-14, however, they have been generalized in a different order. As a result different concepts have been created.

Figure 5-15: Instance order sensitivity in G-tree formation.

The sensitivity of the G-tree structure to the order of F-tree instances is greater when a system is first starting up than it is at a later time. This is because a larger percentage of information is reorganized for each new instance when there are less data in the G-tree. If this initial order sensitivity is a problem in a given domain, two simple solutions can be applied. The first is to have a human "expert" build up the initial G-tree himself. The second would be to have someone choose a particular order of instance hierarchies to feed into the system as a training set.

5.5 Other issues

We have covered most of the major considerations pertinent to the MERGE form of hierarchy generalization. However, there still remain several other issues to investigate. Depending on the domain in which MERGE is used, these vary in

importance. In this section we discuss three concerns that are common to almost any domain, but are less crucial than the ones posited in the previous sections.

5.5.1 Identifying G-tree roots

The presence of several G-trees in a MERGE-based system has been mentioned. Each G-tree serves to classify a different sub-hierarchy in the instance F-trees. The question that naturally arises is which G-trees are needed for a particular domain. This question must be answered in order to create a useful system.

A hierarchy understanding system based on the MERGE scheme needs very little initial information. It is designed to incrementally build its own knowledge base by comparing incoming instances. However, it does need a small initial set of data so that instance hierarchies can be encoded in terms of comparable concepts. For example, CORPORATE-RESEARCHER has the initial concepts of a chairman-of-the-board, president, vice-president, and the like encoded in it. Each new F-tree is specified by temporarily making each memette in it a variant of one of these executive position memettes. When a new instance F-tree is matched against a concept or instance F-tree, embedded in G-trees, these initial concepts serve as a means for comparison -- vice-presidents would match other vice-presidents, presidents would match presidents, etc. CORPORATE-RESEARCHER needs as many initial concepts as there are executive positions in the union (mathematical set union) of all the corporate charts it is expected to understand.

It is important to point out that this matching is only used for a first approximation to get the matching process started. In the case of CORPORATE-RESEARCHER, it is possible to have a president in one company match to a vice-president in another. This would happen, for example, if each corporate officer (i.e., the president in one company, and the vice-president in the other) had a similar sub-hierarchy of members that report to him. (We describe how this type of matching proceeds in more detail below.)

In CORPORATE-RESEARCHER, it is relatively easy to determine the necessary initial concepts. In domains with more complex objects, there can be very many initial concepts. The disc drive patent abstracts that RESEARCHER is intended to understand describe many different physical objects. Consequently, around 200 object concepts are included in its initial knowledge structure. These are all potential parts of a disc drive such as a motor, spindle, read/write head, housing, bearings, etc.

The natural language input to RESEARCHER is not systematized -- there is not a consistent terminology used to describe objects. This necessitates a dictionary that maps several words into a single concept as part of a conceptual analysis system (see [Lebowitz 83b] for an explanation of RESEARCHER's text understanding process). In the process of creating this dictionary (done manually) the initial object concepts (for the G-trees) are created as well.

An initial concept is usually just an empty memette, but it can be more complex, representing anything up to a complete F-tree. The advantage of using well described initial concepts is that the system will have better generalizations for the first few F-trees it processes. The knowledge stored in initial concepts is equivalent to that which would have been gathered from generalizing several instance F-trees. The disadvantage in using large initial concepts is that it creates more work for the human systems builder and forces assumptions about generalized concepts that may be misleading into the system.

5.5.2 Matching F-trees

The generalization process can be broken down into three parts: matching one F-tree against another, locating the correct place in the G-tree using this matching procedure, and incorporating a new concept into the G-tree at the proper location in accordance with the inheritance formalism. In previous sections, we have dealt with the later two of these sub-processes. Obviously, they cannot take place without first having some way of matching two F-trees.

The goal of an algorithm that matches one F-tree against another in MERGE is to produce the "best" correspondence between memettes comprising the trees. In order to achieve the best matching it may be necessary to level-hop so the algorithm chosen must be capable of this function. We have already described level-hopping and will see examples of it in Chapter 6.

The definition of a *best* match must be based on some metric for how similar two memettes are and how important their relative positions in their respective F-trees are. (We will ignore relations among memettes and structure-independent data for the purpose of this discussion, but they must be included in a complete matching algorithm.) There are several heuristics that can be used to derive these criteria, some are domain specific while others are universal to all domains.

Distinguishing levels in the F-tree is an important factor in the matching process. Higher levels should be given more significance than lower levels. This captures the idea that the whole of a hierarchy is more important than any of the sub-hierarchies it encompasses. (Of course, this assumes that a goal of MERGE is to preferentially classify the higher level objects it processes rather than their parts.)

When two memette frames are compared in isolation they should be considered most similar if they are variants of the same parent memette. They are increasingly less similar if their lowest common ancestor is higher in the G-tree in which they are indexed. If they have no common ancestor in the G-tree, then they are least similar. In addition, other memette features (e.g., the contents of the PROPERTIES slot) can also be accounted for when comparing two frames. For example, both objects being made of the same material may be important in some physical object domains.

Organizing these individual heuristics (depth in the F-tree, intersection depth in the G-tree, feature similarities, and others) can be done effectively by using a point assignment grading scheme. (See [Winston 80] for another such scheme, and [Tversky 77] for work on similarity measures.) Using such a scheme, it is possible to match memettes that have no common G-tree ancestor but do have several common F-children. This is largely a pragmatic approach. It is difficult to argue that humans use a mathematical point assignment scheme in comparing objects, but it is common practice in AI work, nonetheless.

When two F-trees are compared, first their leaf memettes are compared and points are assigned for each match of memettes with common ancestors. The memette pairing that gives the highest sum of these points is chosen. Then the parents of these leaf memettes are compared. Their score is found by adding their own match score to the sum of their children's scores. If these parent memettes have no common ancestor in the G-tree, they can still have a high score (and thus a high correlation) if a match of their children results in a high score. (The determination of a criteria for allowing parent nodes to match when enough of their children do is specific to each implementation of MERGE.) This scoring process continues, recursively, up the F-trees. Each successively higher level is weighted more strongly than the levels below it, thus emphasizing the importance of matching an entire hierarchy. A more complete description of this algorithm is given in Appendix A.

The process terminates when the root memette is reached. The final score is then used as the basis for comparing a new instance F-tree against generalized concept F-trees that reside in the G-tree. This is done in order to locate the correct place to index an instance or create a new generalization. The final process of incorporating a new instance into the G-tree is straightforward but messy to program. Inheritance, modified by addition, subtraction, and substitution, must be taken care of. The results of any needed level-hops must be reconciled. Relations, properties, and other data must also be processed, if they have not already been so during the matching process.

5.5.3 Reorganizing memory

The examples shown above demonstrate that memory is continually being reorganized as new instances of hierarchies are brought into MERGE. This is intended to mirror, in some sense, the way humans perform incremental learning. Unfortunately, if instances are fed into a MERGE-based system in varying orders it is possible to create very different knowledge structures -- some may be much "better" than others (according to the F-tree matching metric being used). Humans seem to have some way of recognizing when memory is in need of a gross (large scale) reorganization. They do, on occasion, have an insight as to how things relate to each other and then restructure the way they think about them. This is sometimes called the *aha* response.

MERGE has the capability to reorganize the variant links that comprise the G-tree. This is evident in the way it incrementally reorganizes memory. The problem that remains is how to recognize when a major reorganization is necessary.

One possible way to recognize when reorganization is needed would be to have a system re-process all the instances it has already seen in the context of the existing knowledge base. If few changes to memory take place during this re-processing, it is an indication that memory is well structured. This would be true because each re-processed instance would find the same location to be stored in the G-tree as it did the first pass through the system. Otherwise, "better" places (those that give better F-tree matches) in the G-tree would most likely be found, where the instances could be stored the second time around. (Of course, each instance must be removed from its original place in memory on the second pass -- instances are stored in just one place in memory.) Memory could then be restructured so that it stores each instance -- in the location found during the second pass. The major drawback to this approach is that the instances must be saved in their original form for an indefinite period of time. While this may be possible in some computer implementations, in general it is not a good solution.

Undoubtedly, there are other possible solutions to this problem, but we have not explored them. In Chapter 6, examples will be given demonstrating learning without resorting to a massive reorganization of memory.

5.6 Summary

Generalizing hierarchically structured objects is a difficult task. The major complications stem from three sources: arbitrarily deep F-tree representations, generalizing memettes at each level in the F-trees, and only having data available incrementally as opposed to all at one time. This necessitates a generalization scheme that incorporates the constituent memettes of an instance F-tree into several G-trees. The F-tree sub-hierarchy that each memette is the root of gets classified into a different G-tree. The G-trees are continually modified by small changes needed to incorporate this new information.

Inheritance is used to capture the commonalities and differences among the variants of a concept in a G-tree. Because of potential ambiguities that can arise if a memette is made a variant of more than one concept, inheritance from multiple conflicting sources is not allowed.

The addition, subtraction, and substitution operators modify the inheritance set up by variant links. If an instance F-tree differs from its generalized concept only in one memette at a particular level then all levels above this must necessarily be different. The substitution operator is used to encode this type of variation by allowing all other, non-changed, data to be inherited from the concept memette.

Most descriptions of hierarchies in the real-world are incomplete. The major obstacle posed by incomplete data is that one or more levels may be left out of an F-tree representation. Level-hopping is therefore necessary in a matching algorithm in order to find correspondences between representations. This algorithm is used in locating where in the G-tree an instance F-tree should be stored.

The process of generalizing takes place in three phases: 1- matching a new instance F-tree against those already in memory; 2- locating the "best" place in the G-tree to store this new instance (using the result from the matching process); 3- incorporating the new F-tree into memory using inheritance modified by addition, subtraction, and substitution.

MERGE primarily creates incremental, conjunctive, structure-dependent generalizations. Learning is carried out by making small changes in the knowledge base. On occasion it may be necessary to do a massive reorganization of some parts of memory -- somewhat akin to the human *aha* response.

An ideal MERGE-based system embodies many principles and features that make it useful as an intelligent information system. However, most real systems need only use a subset of them. The basic qualities that a MERGE system must have include: structuring memory in terms of generalizations, dynamically reorganizing these generalizations while incrementally learning, creating parallel G-trees, and the ability to automatically classify many instances in a large domain. We have built two programs that use a better than basic, but less then ideal, implementation of MERGE. CORPORATE-RESEARCHER understands upper-level corporate management hierarchies. RESEARCHER reads and understands patent abstracts about physical objects (disc drives). Both systems add the ability to level-hop to the list of basic features of MERGE. In addition, RESEARCHER uses its knowledge base to assist in processing further input. Detailed demonstrations of these programs are given in this chapter.

6. MERGE - A Scheme for Understanding Hierarchies

6.1 Introduction

So far we have described issues having to do with representing individual hierarchies and using generalization to organize them in memory. In the process, the interaction between representation and generalization has not been examined closely. The MERGE scheme of hierarchy understanding focuses on this interaction by unifying representation and generalization in a way that enhances the functioning of each.

In order to comprehend how MERGE works, it is necessary to fully understand this mutual enhancement. Our scheme is a form of generalization-based memory. It stores representations of individual objects in terms of how they vary from instances and generalized concepts already in memory. Thus, the instantaneous state of memory will affect how an unknown object is perceived (because we are incrementally generalizing); hence, previous generalizations influence the representation of an object. The converse of this statement is self-evident -- the representation of an object influences what generalizations can be made about it. By structuring memory correctly, these ***influences*** can be made beneficial to the system, thus producing the desired effect of ***mutual enhancement*** of both representation and generalization.

Although we have described both parts of this feedback process to some extent (i.e., how representation influences generalization and vice-versa), we have not shown how the entire process operates, nor exactly what is gained through this type of processing. The purpose of this chapter is to do so. Our presentation will begin with a complete description of the ideal MERGE scheme. It will be followed by a description of two systems that use MERGE to understand hierarchies.

A description of the ideal MERGE scheme is given in order to provide the reader with the theoretical underpinnings of this scheme. The representation/generalization feedback cycle is explained in detail along with extensions that make this a practical understanding scheme for hierarchies.

CORPORATE-RESEARCHER and RESEARCHER are hierarchy understanding systems that use MERGE. Each of these is described in an attempt to demonstrate both how MERGE is implemented for a specific application, and that the MERGE scheme can be applied to many hierarchical domains.

CORPORATE-RESEARCHER is used to build a knowledge base that classifies corporate hierarchical structures. It gets its input from corporate charts along with some supplemental relation information from textual descriptions. The data is hand-coded by humans who are not necessarily expert at understanding corporate hierarchies, but who are familiar with the representation formalism. RESEARCHER reads and processes information from patent abstracts that describe complex physical objects [Lebowitz 83b]. (Currently, RESEARCHER reads about computer disc drives and related devices.) The data is automatically parsed into representations of single patents (see [Wasserman and Lebowitz 83] for a full description of the representation scheme). These representations are then incorporated into memory which can then be used to answer questions, classify objects, and help disambiguate further input.

The same basic MERGE scheme is used in both systems. However, CORPORATE-RESEARCHER and RESEARCHER process data in very different hierarchical domains, with different input sources. In addition, these domains have elements of both artificial and naturally occurring hierarchies. The complex physical objects that RESEARCHER reads about are man-made artifacts (e.g., disc drives). Corporate structures are man-made, but seem to take shape naturally. For these reasons, we believe that these systems demonstrate the wide-ranging applicability of MERGE-based understanding.

6.2 MERGE

The basic question to be asked about the MERGE scheme is: exactly how do representation and generalization affect each other, and what is the benefit of their integration? To answer this, we first explain the basic representation/generalization feedback cycle, then some extensions to the basic scheme, and finally we enumerate the features of the ideal MERGE scheme. Throughout this section we will point out the advantages of using a combined representation/generalization scheme as opposed to the more conventional way of treating these as separate processes.

6.2.1 Basic MERGE

The basic representation/generalization feedback cycle in MERGE is best illustrated with an example. Because corporate hierarchies are relatively easily understood (by people) and are obviously hierarchical in form, we will use an example from this domain.

MERGE is generally intended for use in domains with large numbers of instance hierarchies. Therefore, we are usually integrating a new instance representation into a large memory structure, one that already classifies many other instances. Demonstrating all the details of MERGE with such a large memory structure would be extremely complicated, so we will use a simplified example with only three instance hierarchies.

Our sample run of a MERGE-based system (actually CORPORATE-RESEARCHER, but without using program output) starts off with two complete F-trees representing different corporations, as shown in Figure 6-1. The first step in the cycle is the generalization of these F-trees to form G-trees that categorize each element in the representations. Generalizations require the comparison of individual memettes, which is made possible because each member of these two corporations has been defined in terms of an initial concept. The initial concepts that are needed in this example are: ***chairman***, ***president***, ***vice-president***, ***treasurer***, and ***manager***. We have not shown these initial concepts. However, the reader should assume that the names given to each member of these F-trees indicate that they are variants of these initial concepts.

CHAIRMAN-1-1: president-1-1
PRESIDENT-1-1: vice-president-1-1
VICE-PRESIDENT-1-1: manager-1-1

CHAIRMAN-2-1: president-2-1
PRESIDENT-2-1: vice-president-2-1, vice-president-2-2, vice-president-2-3

The top F-tree represents corporation-1, while the bottom figure is for corporation-2. The names given to each memette are indicative of what they are initially variants of. The numbering scheme used in this, and the next few diagrams, first indicates the corporation which the memette is a part of, followed by a unique identification number within each corporation. Thus, ***vice-president-2-3*** is the third vice-president in corporation-2.

Figure 6-1: Two corporate F-trees.

Figure 6-2 shows the unified memory structure after generalizing corporation-1 and

corporation-2. (The naming scheme that we are using is described in the captions of this figure and the previous one.) The generalizations produced serve to indicate the similarities of the two corporations. What they have in common is that the chairman (*chairman-#-1*) has a president (*president-#-1*) followed by a vice-president (*vice-president-#-1*). (Note that *vice-president-1-1* was matched to *vice-president-2-1* as opposed to either of the other two vice-presidents. All three of these vice-presidents match equally well to *vice-president-1-1*, thus the program made an arbitrary choice.) So far, we have seen no additional benefits (other than generalization of hierarchies) from the use of MERGE. The next step will exemplify the feedback process.

CHAIRMAN-#-1:	president-#-1, >chairman-1-1, >chairman-2-1
CHAIRMAN-1-1:	president-#-1–+president-1-1
CHAIRMAN-2-1:	president-#-1–+president-2-1
PRESIDENT-#-1:	vice-president-#-1, >president-1-1, >president-2-1
PRESIDENT-1-1:	vice-president-#-1–+vice-president-1-1
PRESIDENT-2-1:	+vice-president-2-2, +vice-president-2-3
VICE-PRESIDENT-1-1:	+manager-1-1

This is the memory structure formed by the generalization of the two F-trees diagrammed in Figure 6-1. The generalized concepts that have been created are given "#" symbols to indicate that they are not part of an instance corporation. The number after the "#" symbol is a sequence number to allow for multiple generalized concepts.

Figure 6-2: Unified memory structure for two corporations.

Consider the incomplete F-tree shown in Figure 6-3. It represents a corporate structure with two vice-presidents (*vice-president-3-1* and *vice-president-3-2*) and a treasurer (*treasurer-3-1*) that report to some unspecified intermediary (*x-3-1*). One can imagine that this description came from a mangled corporate chart or, more likely, from a natural language description that failed to mention what position this intermediary holds. The hypothetical MERGE-based system tries to incorporate this new F-tree into its unified memory structure. Using an algorithm similar to the one described in Sections 5.4 and 5.5, the system finds that corporation-3's structure most closely matches that of corporation-2, even though it must match *x-3-1* against *president-2*.

At this point, the system is about to take its first "intelligent" step. It will make *x-3-1* a variant of a president when it builds the F-tree into its knowledge structure. Thus, it will use the results of a previous generalization to enhance a new representation. The generalization is that "all corporations (seen so far) have a chairman, president, and at least one vice-president." Since corporation-3 explicitly has the first and last memettes in this chain of three, and the middle member is

CHAIRMAN-3-1: x-3-1
X-3-1: vice-president-3-1, vice-president-3-2, treasurer-3-1

Corporation-3's F-tree is depicted here. *X-3-1* is known to exist, but it is not known what position it corresponds with (e.g., chairman, president, vice-president, etc.)

Figure 6-3: Corporation-3's representation.

undefined, it fits this generalization if *x-3-1* is a president. The MERGE-based system has used inheritance (modified by addition and substitution) from a concept it has created to fill in missing information. Figure 6-4 shows the resulting unified memory structure, which assumes *x-3-1* is a president.

CHAIRMAN-#-1: president-#-1, >chairman-#-2, >chairman-1-1
CHAIRMAN-#-2: president-#-1–+president-#-2, >chairman-2-1, >chairman-3-1
CHAIRMAN-1-1: president-#-1–+president-1-1
CHAIRMAN-2-1: president-#-2–+president-2-1
CHAIRMAN-3-1: president-#-2–+x-3-1
PRESIDENT-#-1: vice-president-#-1, >president-#-2, >president-1-1
PRESIDENT-#-2: +vice-president-#-2, >president-2-1, >x-3-1
PRESIDENT-1-1: vice-president-#-1–+vice-president-1-1
PRESIDENT-2-1: +vice-president-2-3
X-3-1 (president): +treasurer-3-1
VICE-PRESIDENT-1-1: +manager-1-1

Corporation-3 has been generalized into memory. During the process it was found that *x-3-1* should be a variant of a president. Its incorporation into memory has caused new concepts to be built. *President-#-2* and *chairman-#-2* represent a corporation with two vice-presidents.

Figure 6-4: Unified memory structure for three corporations.

The next step, completing the feedback cycle, has already been taken by the formation of the most recent generalization. *Chairman-#-2* represents a corporation with a chairman, president, and two vice-presidents. Thus, the concept of a corporation with two vice-presidents has been created. Corporation-3's representation was responsible for the need to make this generalization. Hence, the representation (of a single corporation) has enhanced the generalization structure. The next company's F-tree, to be incorporated into this unified memory structure, will now have more concepts to be compared against. It will have a better chance

of having any missing or ambiguous information filled in (i.e., it has a better chance of being understood more completely). It, in turn, may contribute to the concepts formed by the system, allowing for even more improvement in the system's ability to understand future input. Each run of this generalization/representation cycle has the potential to enhance the system's knowledge organization while at the same time improving the representation of the latest instance.

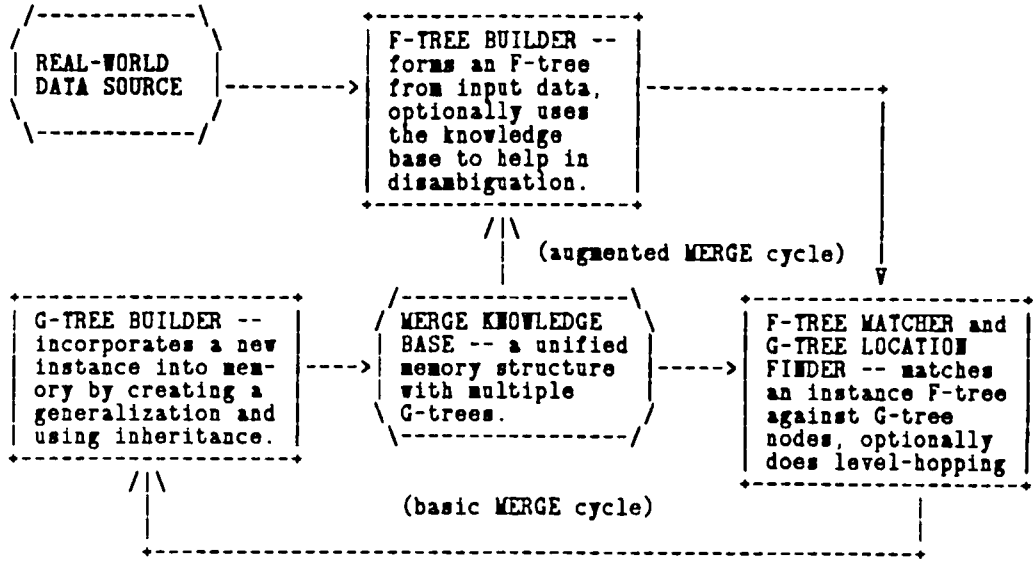
The integration of representation with generalization has several advantages that could not have been achieved otherwise. First, there would have been no way to determine that *x-3-1* should be a president unless corporation-3's F-tree was compared against other F-trees. Second, if *x-3-1* were only compared against isolated F-trees it would be hard to have confidence in believing that it is a president. By making comparisons against generalized concepts this confidence level is increased. Third, once the proper (i.e., best) match is found, the F-tree for corporation-3 can be modified accordingly. This newly modified F-tree is then incorporated into the appropriate G-trees for use in later processing. If it were kept separate from other instance F-trees, it would be of little help in processing future input.

From this example we can conclude the following: integrating representation with generalization can be used as the basis of a useful understanding system. The feedback that this relationship permits enhances both the representation of individual hierarchies and generalizations built from these representations. This enhancement is made possible because an object's representation is intertwined with the generalizations that it is included in, and that generalizations are automatically created when a new instance representation is incorporated into memory.

6.2.2 The MERGE cycle

The *basic MERGE cycle* demonstrated above can be viewed as a three stage process as diagrammed in Figure 6-5. The basic cycle proceeds as follows: 1- a new instance F-tree is created from an input data source, 2- this F-tree is compared against concepts in memory by searching down a G-tree (of the top-level F-tree memette) until a "best" match is found, 3- this F-tree is incorporated into the knowledge base by creating a generalized concept (if needed) at the location that gives the best match.

Two common enhancements to this basic cycle are also shown in Figure 6-5. The *augmented MERGE cycle* is used to help the F-tree builder to form correct representations. Information from the knowledge base can be used to disambiguate input data, as is done in RESEARCHER when it parses patent abstracts. The other enhancement is level-hopping (described in Section 5.2) that is done during the matching of one F-tree to another.



A block diagram of the major components of a MERGE-based system are shown. The *basic MERGE cycle* is an essential part of MERGE. The *augmented MERGE cycle* is used when programs need to make use of the knowledge base to assist in processing further input.

Figure 6-5: The MERGE cycle.

6.2.3 Beyond basic MERGE

The example given in section 6.2.1 demonstrates only the essential parts of an ideal MERGE-based system. There are many other features that are necessary for a real-world application of MERGE. We will discuss several of these that are common to all hierarchical domains.

We start by noticing that *x-3-1* in corporation-3 was stated to be "unspecified" in the input. It seems more likely that it would be missing entirely (i.e., the description of corporation-3 completely neglected to mention it). If this were the case, the matching algorithm in MERGE must be able to level-hop in order to find the *chairman-2-1/chairman-3-1*, *vice-president-2-1/vice-president-3-1*, and *vice-president-2-2/vice-president-3-2* correspondences. Level-hopping is an important feature of MERGE, particularly when used in domains with many incomplete descriptions of hierarchies.

MERGE represents a level-hop by using a *null memette*. A null memette is one that serves a dual purpose. It acts as a place holder for an F-child that has been specified in an instance F-tree. In addition, it can also act as nothing. That is, it represents a memette that should really not be there at all. To determine which of these roles a null memette is serving, an extra slot is added to a memette frame. We call this the *ALTERNATE-VARIANT-OF* slot, indicating that it is much like the VARIANT-OF slot.

The ALTERNATE-VARIANT-OF slot is in some sense a means for inheriting non-conflicting data from two sources. If the ALTERNATE-VARIANT-OF slot of a particular memette is filled, then the memette will inherit its identity (i.e., what it is an instance of) from the parent memette that is indicated in the ALTERNATE-VARIANT-OF slot. However, it will inherit any F-children (i.e., its structural information) from the NULL# memette that fills the VARIANT-OF slot. This does not violate any of the constraints that were mentioned in Section 5.3 because the data inherited from these sources is of a completely different nature. The ALTERNATE-VARIANT-OF slot only allows for structure-independent data to be inherited while the VARIANT-OF slot is used to allow for inheritance of structure-dependent data.

Figures 6-6(a) and 6-6(b) show two simple F-trees. The names given to each memette indicate what they are initially variants of. When they are generalized, a level-hop is required in order to find the correspondence of *vice-president-1* to *vice-president-2*. The resulting unified memory structure is shown in Figure 6-6(c). A null memette (*null-#*) has been inserted between *chairman-#* and *vice-president-#* in the generalized concept. Note that *chairman-#* has both *chairman-1* and *chairman-2* as variants and that *chairman-2's* F-tree is exactly the same as the generalized F-tree. In order for *president-1* to inherit a vice-president from the generalized F-tree, it is made a variant of *null-#*. Unfortunately, it will lose its identity (i.e., being a variant of a president) unless its original VARIANT-OF link is kept somewhere. The ALTERNATE-VARIANT-OF slot serves this purpose.

A memette which is a variant of a null memette represents nothing if its ALTERNATE-VARIANT-OF slot is empty. Thus, *chairman-2* still represents a corporate structure with no president since it inherits a memette that is a variant of *null-#* (actually, it is a copy of *null-#*) and that memette has an empty ALTERNATE-VARIANT-OF slot.

As was mentioned in previous chapters, a MERGE-based system should ideally represent and generalize about other information, aside from the F-children of a hierarchy. Most hierarchical domains have at least some structure-dependent data (relations) as well as some structure-independent data (e.g., properties). Both of these forms of data are included in the ideal MERGE scheme.

Relations that are absent in the representation of an instance hierarchy can be inferred from generalized concepts of similar hierarchies that have them. The feedback cycle processes relations in the same way as it does F-children. Representations of relations are enhanced by previously made generalizations of similar relations, and these newly encoded representations are then used to form more extensive and detailed generalizations.

For example, if corporation-1 and corporation-2 (in Figures 6-1 through 6-4) each

CHAIRMAN-1: president-1
PRESIDENT-1: vice-president-1
VICE-PRESIDENT-1:
(a)

CHAIRMAN-2: vice-president-2
VICE-PRESIDENT-2:
(b)

CHAIRMAN-#: null-#, >chairman-1, >chairman-2
CHAIRMAN-1: null-#-+president-1
CHAIRMAN-2:
NULL-#: vice-president-#, >president-1
PRESIDENT-1: (ALTERNATE-VARIANT-OF president)
(c)

Diagrams (a) and (b) are two corporate F-trees. The unified memory structure shown in diagram (c) is the generalization of these two F-trees. A level-hop is required so that the best match is found. Consequently the ALTERNATE-VARIANT-OF slot is needed so that *president-1* can retain its original identity.

Figure 6-6: The ALTERNATE-VARIANT-OF slot.

have the relation that the president meets frequently with the chairman, then the generalized concept of a corporation (headed by *chairman-#-1*) would have the relation MEET-OFTEN(*president-#-1*, *chairman-#-1*). Assuming that the description of corporation-3 said something about meetings among the top-level executives, but did not specify whom, MERGE would assume that the participants are *x-3-1* and *chairman-3-1*. Similarly, corporation-3 might contribute to the relation information stored in the *chairman-#-2* concept. Whatever relations corporation-2 and corporation-3 have in common (or their common relation parts) would be captured.

Structure-independent data (properties) are treated in much the same way. Their processing is somewhat easier because it can be done without regard to how the remainder of the F-trees match (i.e., structure-independent data is local to a memette). This is not so for relations. In the example above, MERGE first had to know that *x-3-1* matched with *president-2-1* before it could assume any of the relation data that pertains to *x-3-1*. Thus, MERGE must be able to process relations in conjunction with F-children processing.

A few other extensions to MERGE are also possible in an idealized implementation. These ideas have not been implemented in either CORPORATE-RESEARCHER or RESEARCHER. We present them as suggestions for future research.

Most of the issues that were discussed in Chapter 5 could be applied in an idealized MERGE-based system. The capability to do a massive reorganization of memory when it becomes too "unreasonable" (i.e., too many generalized concepts that do not give a good idea of the known data have been created) would be a useful feature of such a system, although it may not be necessary. Allowing information to be inherited from multiple sources would be desirable in many applications. Unfortunately, we know that if there is a possibility for conflicting data to be inherited then the disambiguation of this data becomes an intractable problem. However, non-conflicting data can be inherited from multiple sources.

It is also possible to extend the basic MERGE scheme along the lines of how it represents data. We have stated that it is difficult to create a complete relation representation scheme for some domains (although a fairly good partial scheme is not too difficult to develop). The ideal MERGE-based system would have some mechanism for dynamically altering the way it encodes relation characteristics.

6.2.4 Ideal MERGE

Here, we have described the features that an ideal MERGE-based system would have. We are about to study two examples of hierarchy understanding systems that use MERGE, but before doing so a summary of what the ideal MERGE scheme includes is in order. Neither CORPORATE-RESEARCHER nor RESEARCHER is ideal; there are gaps in each system that will be described in Section 6.5. By enumerating the features of an ideal system we give the specifications that another researcher would need to build a MERGE-based system, while at the same time providing a comparison standard for our own implementations.

The following list of features is numbered for reference purposes (see Section 6.5).

Features of MERGE:

1. Generalization-based memory - Generalizations are used as the basis for large scale memory organization, with instances stored in terms of generalizations.

2. Dynamic memory - Memory is continually reorganized in small sections at a time by changing old generalizations and creating new ones.

3. Framed-based representations - Memettes are used to describe both real and generalized objects. A memette structure can represent any level of detail in an object.

4. Parallel generalizations - Multiple G-trees exist in parallel. They each are a knowledge structure that categorizes a different object.

5. Inheritance - Inheritance allows for easily recognized similarities and/or differences among instances and generalized concepts of hierarchies.

6. Automatic classification - The G-trees built by MERGE serve as a way of automatically categorizing instance hierarchies.

7. Incremental learning - Incremental generalization means that a system learns each time a new instance is presented. Thus, a MERGE-based system continuously tracks data from the real world.

8. Large domains - MERGE is designed to process a very large number of instances. It is, therefore, suitable for understanding domains that would be difficult for a human to grasp.

9. Massive reorganization/error correcting - The memory structure should be reorganized when it is found to become unwieldy, or bad representations have led to incorrect generalizations to be made. That is, the G-trees become awkward and don't seem to be a good classification of the instances fed into the system.

10. Process varied data - Relations and properties of almost any type can be included in object descriptions. Relation characteristic encodings can be complex or simple. The relation scheme may even use dynamic primitives to represent characteristics.

11. Multiple inheritance - Information can be inherited from multiple sources if the data cannot become contradictory.

12. Level-hopping - a MERGE-based system can deal with incomplete representations or non-standardized hierarchies by using its level-hopping mechanism.

13. Accessible knowledge structure - The unified memory structure in MERGE is accessible for use in other parts of an understanding system. For example, a natural language processing system can use the knowledge base in MERGE to help disambiguate future input.

6.3 CORPORATE-RESEARCHER

A concrete example of a MERGE-based hierarchy understanding system is the best way to demonstrate how a real implementation of MERGE works. To this end, we describe CORPORATE-RESEARCHER, a program that automatically categorizes representations of hierarchical corporate organizations.

We choose to describe CORPORATE-RESEARCHER before describing

RESEARCHER, a more sophisticated understanding system, because CORPORATE-RESEARCHER's domain is more clear-cut. Upper-level corporate structures are obvious hierarchies with well defined F-children (at least on paper). In addition, there are usually only a small number of supplemental relations overlayed on the F-trees. This permits us to focus the discussion on the basic MERGE cycle. The input to CORPORATE-RESEARCHER comes from complete F-tree representations. They are created by hand and have few inconsistencies or omissions.

We will show the basic MERGE scheme, how level-hopping appears, simple relation generalization, and how the theories we have presented tie in with an application. CORPORATE-RESEARCHER was primarily built for the purpose of experimenting with generalization techniques and demonstrating MERGE.

With this in mind, we begin by explaining the basics of the domain and how hierarchies are represented. Following this, we demonstrate an actual run of CORPORATE-RESEARCHER. We then evaluate the program's performance. Throughout this section we use examples from real corporate charts as well as some smaller, hypothetical ones.

6.3.1 The corporate chart

A corporation prepares a chart of its structure for a number of reasons [Webber 75] including: to help in reorganizing a company, to inform employees and outsiders about a company's general structure, to improve channels of communication among employees, to establish reporting pathways, to establish authority and responsibility, and to help in solving internal corporate problems. Regardless of the reason that a company goes through the process of "charting" itself, they are always concerned with two fundamental concepts: ***chain-of-command*** and ***span-of-control***. Chain-of-command is the pathway through which responsibility is passed. Span-of-control refers to the number of subordinates that a superior supervises.

The F-rel that captures the idea of chain-of-command we call REPORTS-TO. The concept of span-of-control is implicit in any F-tree that represents a corporation, but is of primary importance when studying corporate charts. One of the key factors in a company's structure is span-of-control versus the length of chain-of-command. Basic business theory emphasizes the need to keep these in balance [Webber 75].

Most corporate charts are very straightforward, showing only chain-of-command F-rel links. However, there is often other information of interest shown in corporate charts. The most obvious of this is the grouping of F-children according to some specialization criteria. These criteria can be based on geographic location, function, products, and other aspects of a division or office within a company. For example, a president might have a total of ten vice-presidents reporting to him, which are broken into five groups of two. Each group corresponds to a different region in

the United States (e.g., Northeast, South, West, Central, and possessions and territories). Five REGION-GROUP relations could be used to augment a company's F-tree in order to capture this information.

Other relations among arbitrary corporate members appear scattered about in some charts. These include relations such as: ADVISES (e.g., a consultant advises the president), and SUBSIDIARY (e.g., one division is a subsidiary of another, even though they both report to the same person). Certainly there are many other relations within any real corporation. But companies are unlikely to put some of these on their charts. IS-THE-BROTHER-IN-LAW-OF and DOESN'T-SPEAK-TO would not be favorites of the stockholders.

For simplicity's sake, we have chosen to use only a handful of relations in our representations. There are four relations (ADVISES, REGION-GROUP, DIVISION-GROUP, and SUBSIDIARY) that are used along with the REPORTS-TO F-rel. We find that these are sufficient to demonstrate the salient features of MERGE in CORPORATE-RESEARCHER. However, other relations are very easy to add to the system.

The source of corporate charts that are used here is a compendium found in [White 63]. They have been encoded by hand into the memette-based F-trees that CORPORATE-RESEARCHER uses. In some cases they have been edited. This editing was necessary for two reasons. The first is because corporate charts are sometimes specified at very different levels of detail. That is, larger corporations tend to show only their upper-level management positions, while small companies describe their structures down to the blue-collar level. The second reason is simply to keep the sizes of representations practical, as they are being hand-coded.

Throughout this section, and the next, we will refrain from using our compact notation for unified memory structures. Instead, we will use tree diagrams that are automatically output from the program. Both F-tree representations and G-trees are presented. Unfortunately, there is a loss of information in using tree diagrams over displaying the complete unified memory structure. Specifically, the substitution, subtraction, and addition operations can not be seen. However, the end results of using these operations are apparent through inheritance, which is shown.

6.3.2 Some real examples

Figure 6-7 shows the F-tree for the upper-level management of the Lockheed Corporation, as it appeared in the 1950's. We use this as an example to explain CORPORATE-RESEARCHER's notation that will used in the remainder of this section.

The F-tree uses memettes to identify the nodes in the tree. At the right is a cross reference to the name given to each memette that represents a node in the corporate chart. The letter "A" that occurs along some of the F-rel links refers to a relation described in the bottom diagram. It corresponds to the relation, &REL0.

Figure 6-7: Lockheed Corporation's F-tree.

The most obvious difference between this F-tree and a real corporate chart is that numbers appear as the F-children instead of the name of the division, group, or person in the company. These numbers are given to each memette frame so that they can be uniquely identified. We will use the prefix "&MEM" when discussing individual memette frames. Along the right hand side of Figure 6-7 is a list of the memette frames used in the F-tree, followed by the name of the memette. This name is what actually appears on the company's chart. The only other feature to note, is the letter "A" that appears along some F-rel links. It indicates that the F-child is involved in relation "A". The bottom of this figure shows that relation "A" (&REL0) is a DIVISION-GROUP relation where &MEM25 (the president) heads a group composed of &MEM23, &MEM26, &MEM27, &MEM33, &MEM34, and &MEM35.

Figure 6-8 provides information to supplement this F-tree. A description of all the memette frames in the F-tree is shown, here. The column labeled *Function* is a classification of the function that that F-child (subordinate) serves in the corporation. (It is NIL if it was not specified in the company's chart.) We include functions here only to give the reader some more information as to what the real chart looks like. At present, CORPORATE-RESEARCHER does not use this data,

although it could be used in the generalization process; it is an example of structure-independent data, and therefore is easily generalized. The *Variant-Of* column shows the G-tree parent of each memette frame. As was stated earlier, each memette must be a variant of some *initial* memette so that it can be compared against other memettes during generalization. The names given to each of the initial memettes are self-explanatory.

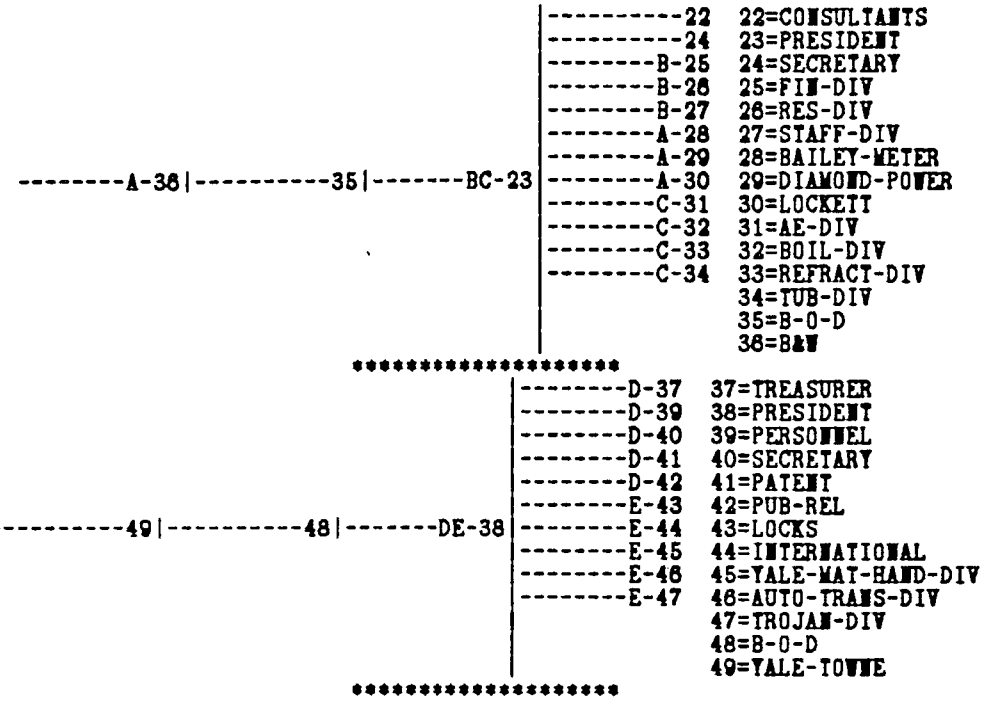
Memette:	Name:	Function:	Variant-Of:
&MEM22	(CONTROLLER)	CONTROLLER	CONTROLLER#
&MEM23	(FINANCE)	FINANCE	V-PRES#
&MEM24	(TREASURER)	TREASURER	TREASURER#
&MEM25	(PRESIDENT)	NIL	PRES#
&MEM26	(COUNSEL)	LEGAL	ATTORNEY#
&MEM27	(DEVELOPMENT)	DEVELOPMENT	V-PRES#
&MEM28	(MANUFACTURING)	MANUFACTURING	V-PRES#
&MEM29	(EXEC-VP)	NIL	V-PRES#
&MEM30	(MARKETING)	MARKETING	V-PRES#
&MEM31	(GROUP-VP-1)	OPERATIONS	V-PRES#
&MEM32	(GROUP-VP-2)	OPERATIONS	V-PRES#
&MEM33	(ENGINEERING)	ENGINEERING	V-PRES#
&MEM34	(PUBLIC-RELATIONS)	PUBLIC-RELATIONS	V-PRES#
&MEM35	(ADMINISTRATION)	ADMINISTRATION	V-PRES#
&MEM36	(C-O-B)	NIL	CHAIRMAN#
&MEM37	(SEN-VP-1)	NIL	V-PRES#
&MEM38	(PLANNING)	PLANNING	DIRECTOR#
&MEM39	(SEN-VP-2)	NIL	V-PRES#
&MEM40	(DIVERSIFICATION-DIRECTOR)	PLANNING	DIRECTOR#
&MEM41	(LOCKHEED)	NIL	CORP#

This table describes the memette frames used in the F-tree (shown in Figure 6-7) in more detail.

Figure 6-8: Details for the Lockheed F-tree

The one remaining point to be made about the F-trees used in CORPORATE-RESEARCHER is the top-level memette. Notice that in this case it is named "LOCKHEED" and is a variant of CORP# Although the C-O-B (chairman-of-the-board) doesn't REPORT-TO the corporation as a whole (as the president REPORTS-TO the C-O-B), we need it in order to get a handle on the ephemeral entity called a corporation. (It also makes it easy to talk about the F-trees since they have meaningful names.)

Two more F-trees, taken from the charts of the Babcock & Wilcox Company (&MEM36) and from the Yale & Towne Manufacturing Company (&MEM49), are shown in Figure 6-9. (The list of relation descriptions appears in Figure 6-10.) They are somewhat simpler hierarchies than the previous one, and will be used to demonstrate CORPORATE-RESEARCHER's generalization process.



Memette:	Name:	Function:	Variant-Of:
&MEM22	(CONSULTANTS)	NIL	CONSULTANT#
&MEM23	(PRESIDENT)	NIL	PRES#
&MEM24	(SECRETARY)	NIL	CORP-SEC#
&MEM25	(FIN-DIV)	FINANCE	V-PRES#
&MEM26	(RES-DIV)	RESEARCH	V-PRES#
&MEM27	(STAFF-DIV)	ADMINISTRATION	V-PRES#
&MEM28	(BAILEY-METER)	NIL	PRES#
&MEM29	(DIAMOND-POWER)	NIL	PRES#
&MEM30	(LOCKETT)	NIL	PRES#
&MEM31	(AE-DIV)	OPERATIONS	V-PRES#
&MEM32	(BOIL-DIV)	OPERATIONS	V-PRES#
&MEM33	(REFRACT-DIV)	OPERATIONS	V-PRES#
&MEM34	(TUB-DIV)	OPERATIONS	V-PRES#
&MEM35	(B-O-D)	NIL	B-O-D#
&MEM36	(B&W)	NIL	CORP#
&MEM37	(TREASURER)	TREASURER	TREASURER#
&MEM38	(PRESIDENT)	NIL	PRES#
&MEM39	(PERSONNEL)	PERSONNEL	V-PRES#
&MEM40	(SECRETARY)	ADMINISTRATION	CORP-SEC#
&MEM41	(PATENT)	LEGAL	ATTORNEY#
&MEM42	(PUB-REL)	PUBLIC-RELATIONS	DIRECTOR#
&MEM43	(LOCKS)	OPERATIONS	V-PRES#
&MEM44	(INTERNATIONAL)	NIL	V-PRES#
&MEM45	(YALE-MAT-HAND-DIV)	OPERATIONS	V-PRES#
&MEM46	(AUTO-TRANS-DIV)	OPERATIONS	MANAGER#
&MEM47	(TROJAN-DIV)	OPERATIONS	MANAGER#
&MEM48	(B-O-D)	NIL	B-O-D#
&MEM49	(YALE-TOWNE)	NIL	CORP#

The F-trees for the Babcock & Wilcox Company (&MEM36) and Yale & Towne Manufacturing Company (&MEM49) appear at the top of this figure. Below them is the description of their memette frames.

Figure 6-9: Babcock & Wilcox and Yale & Towne corporations.

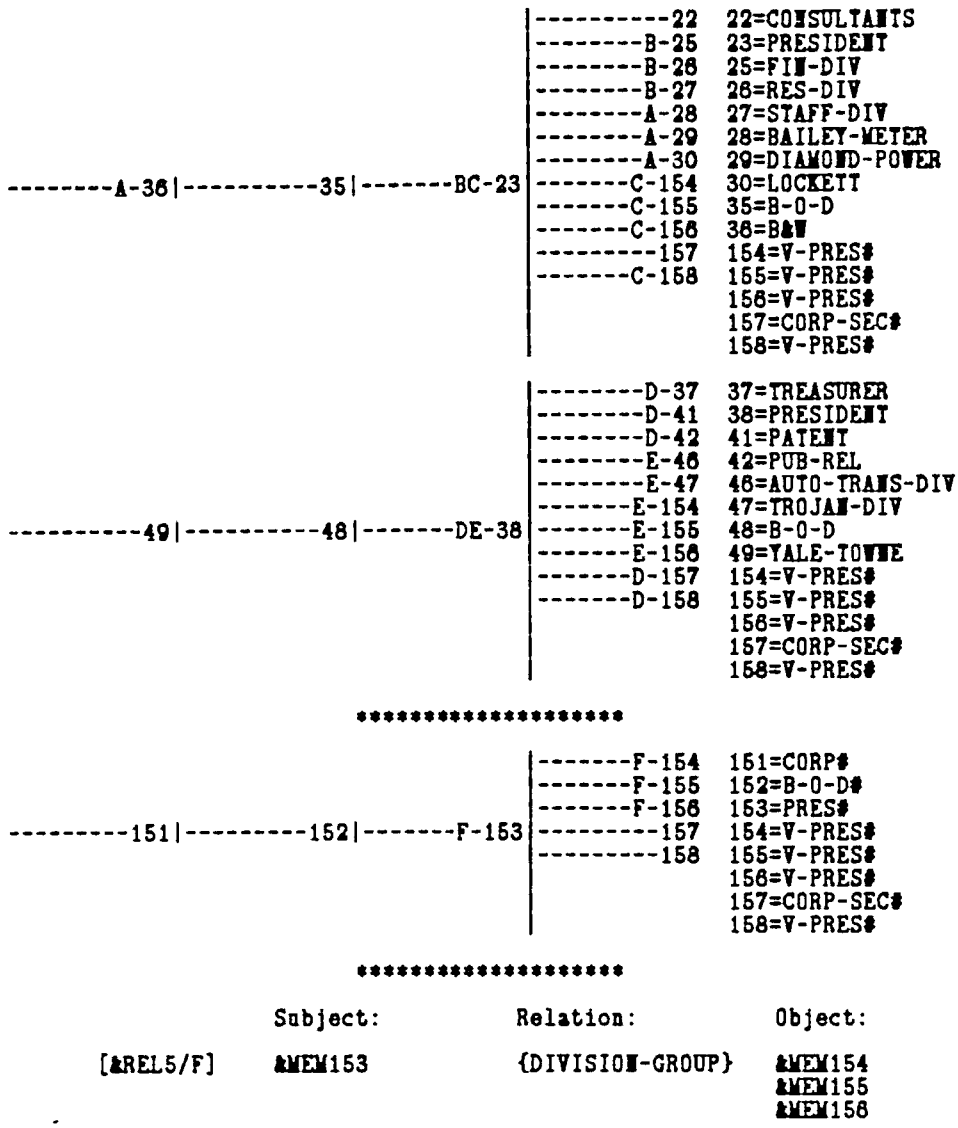
	Subject:	Relation:	Object:
[&REL0/A]	&MEM36	{SUBSIDIARY}	&MEM30 &MEM29 &MEM28
[&REL1/B]	&MEM23	{DIVISION-GROUP}	&MEM27 &MEM26 &MEM25
[&REL2/C]	&MEM23	{DIVISION-GROUP}	&MEM31 &MEM32 &MEM33 &MEM34
[&REL3/D]	&MEM38	{DIVISION-GROUP}	&MEM42 &MEM41 &MEM40 &MEM39 &MEM37
[&REL4/E]	&MEM38	{DIVISION-GROUP}	&MEM47 &MEM46 &MEM45 &MEM44 &MEM43

These relations (A - E) are used in the F-trees for Babcock & Wilcox and Yale & Towne (A, B, and C in Babcock & Wilcox).

Figure 6-10: Relations for Figure 6-9.

During the generalization process, the memettes in the F-tree for Yale & Towne will be matched against those in the F-tree for Babcock & Wilcox. The two basic factors that determine which memettes will be matched are: 1- what fills the VARIANT-OF slot (e.g., is the memette a vice-president (V-PRES#), etc.), 2- what, if any, relations are the memettes involved in. The more heavily weighted of these two factors is the first. If there is more than one match that gives equivalent results then the relations that the memettes are involved in are considered. (The actual algorithm is more complex, but this is correct for this example.) Memettes in similar relations will match better than memettes in different relations, or not in any relations at all.

Figure 6-11 shows the resultant F-trees after generalization. The F-tree in the middle figure represents the concept of a corporation created by generalizing the two F-trees in Figure 6-9. This seems to correlate well with our intuition of what a typical corporation looks like. It has a chairman, followed by a president, followed by some number of vice-presidents (four, in this case) and a corporate secretary. There is also a generalized relation among three of the vice-presidents and the president. The bottom part of Figure 6-11 shows that relation "F" is a DIVISION-GROUP relation and was generalized from the "C" and "E" relations found in Babcock & Wilcox and Yale & Towne, respectively. This is apparent from the fact that memettes &MEM154, &MEM155, and &MEM156 have been inherited from the generalized concept (&MEM153) and that relations "C" and "E" apply to this set of memettes.



The top figures show the F-trees for Babcock & Wilcox and Yale & Towne after they have been generalized. It includes the F-children that are inherited from the generalized corporation shown in the middle figure. Relation "F", shown in the bottom figure, is formed during the generalization process and belongs to the generalized corporation's F-tree (&MEM151).

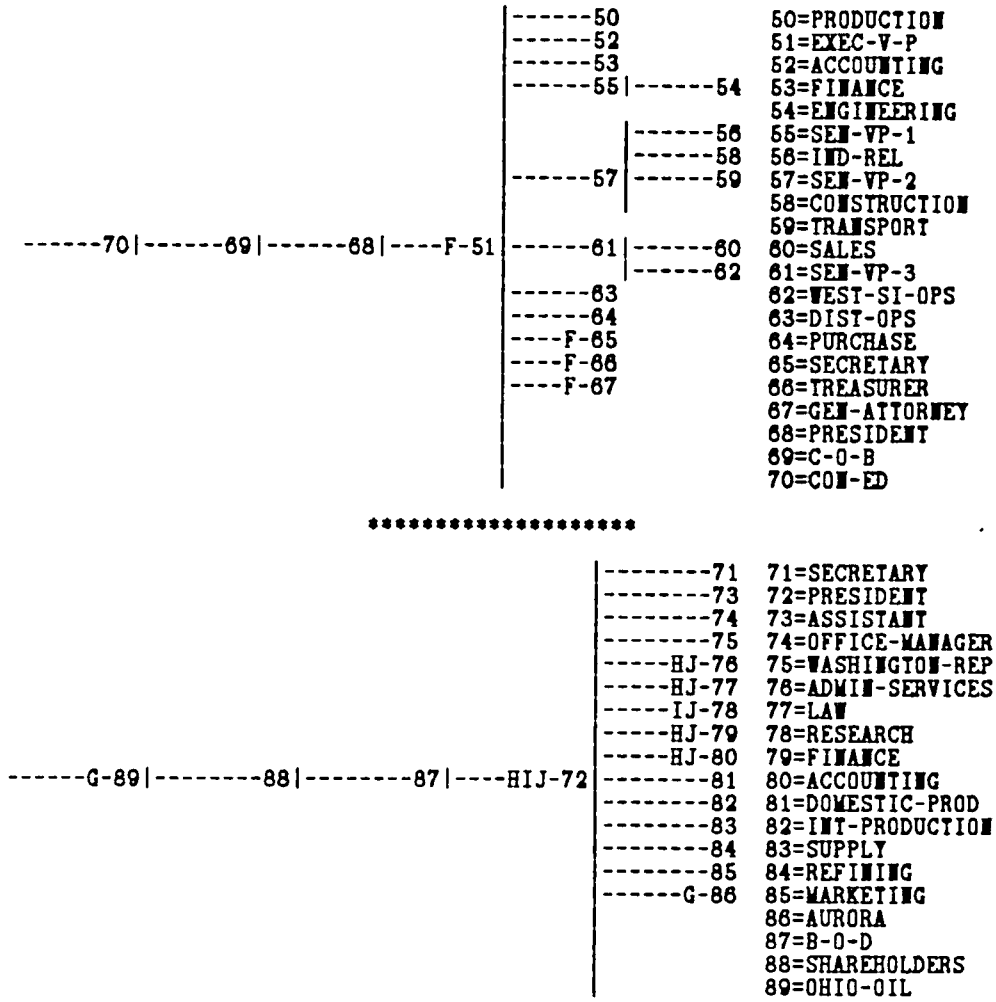
Figure 6-11: Stereotypical corporation.

The main feature of the F-trees shown in Figure 6-11 is the inheritance of generalized F-children. For example, Babcock & Wilcox's F-tree (&MEM36) now has the same structure as it did initially, but inherits four vice-presidents (V-PRES#) and a corporate secretary (CORP-SEC#) from the generalized concept of a corporation (&MEM151) instead of being re-represented. This captures the idea that Babcock & Wilcox's corporate structure is similar to the generalized concept of a corporation but adds another seven subordinates to the president's span-of-control.

So far we have only seen the generalized F-tree. We have not looked at the

underlying G-trees. There must be a G-tree, for instance, that represents &MEM23 and &MEM38 as variants of &MEM153 for this inheritance to take place. There is, but it's not too interesting. To add more interest to the G-trees, two more F-trees will be incrementally generalized into the unified memory structure.

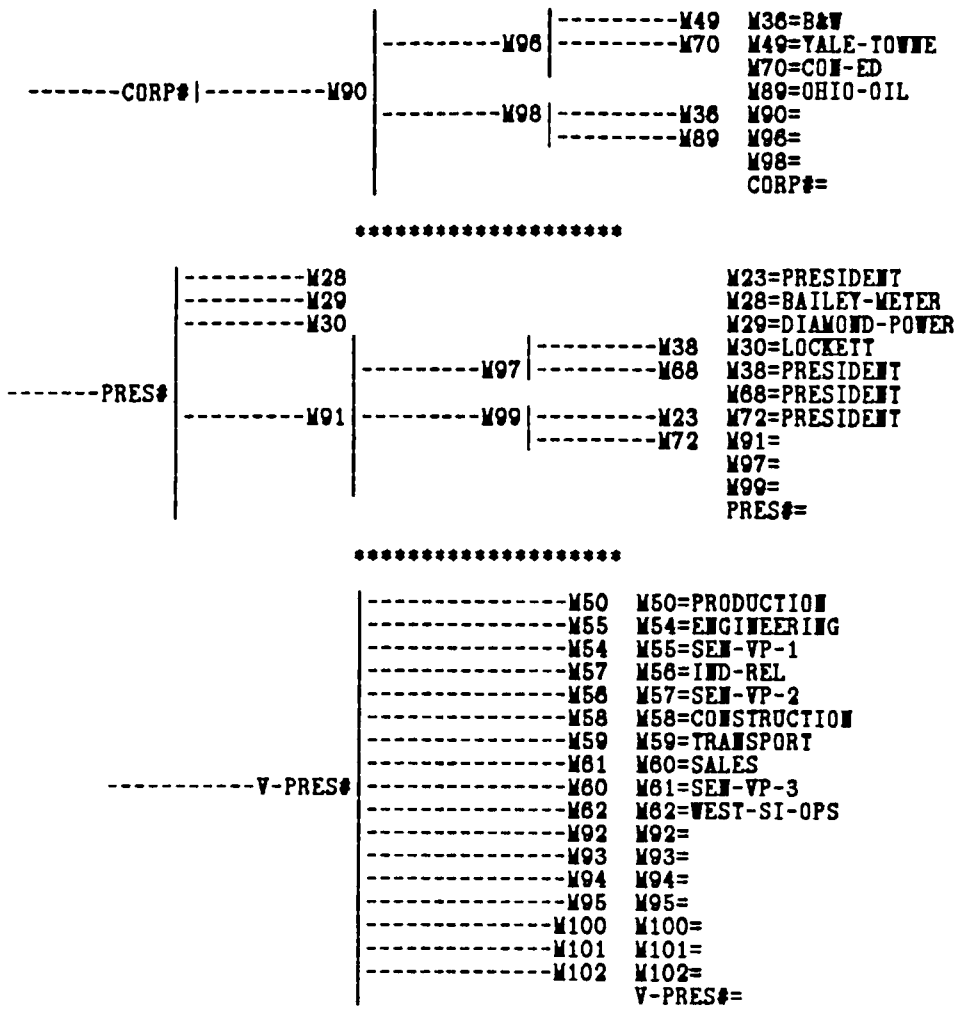
Figure 6-12 shows F-trees for Con Edison (&MEM70) and Ohio Oil (&MEM89). There is nothing new to take note of here.



Two more F-trees are shown here, Con Edison (&MEM70) and Ohio Oil (&MEM89). They should be considered to be in sequence (consecutive memette frame numbers) after the two F-trees shown in Figure 6-9.

Figure 6-12: Con Edison and Ohio Oil corporations.

Figure 6-13 shows the G-trees (after incrementally generalizing all four F-trees) for the concept of a corporation (CORP#), a president (PRES-#), and a vice-president (V-PRES#). G-trees appear similar in form to F-trees in the output of the program, but can be easily distinguished by noting that an "M" prefixes the memette number in the G-trees. The corporation G-tree shows that Yale & Towne (&MEM49) is most similar to Con Edison (&MEM70), and that Babcock & Wilcox (&MEM36) is most similar to Ohio Oil (&MEM89).



All four corporations (Babcock & Wilcox, Yale & Towne, Con Edison, and Ohio Oil) have been incrementally generalized (in this order). These three G-trees show the result of these generalizations. The top G-tree shows how the corporations, as a whole, have been categorized. The middle figure shows how the presidents have been categorized. The vice-president G-tree appears at the bottom of this figure.

Figure 6-13: G-trees for four corporations.

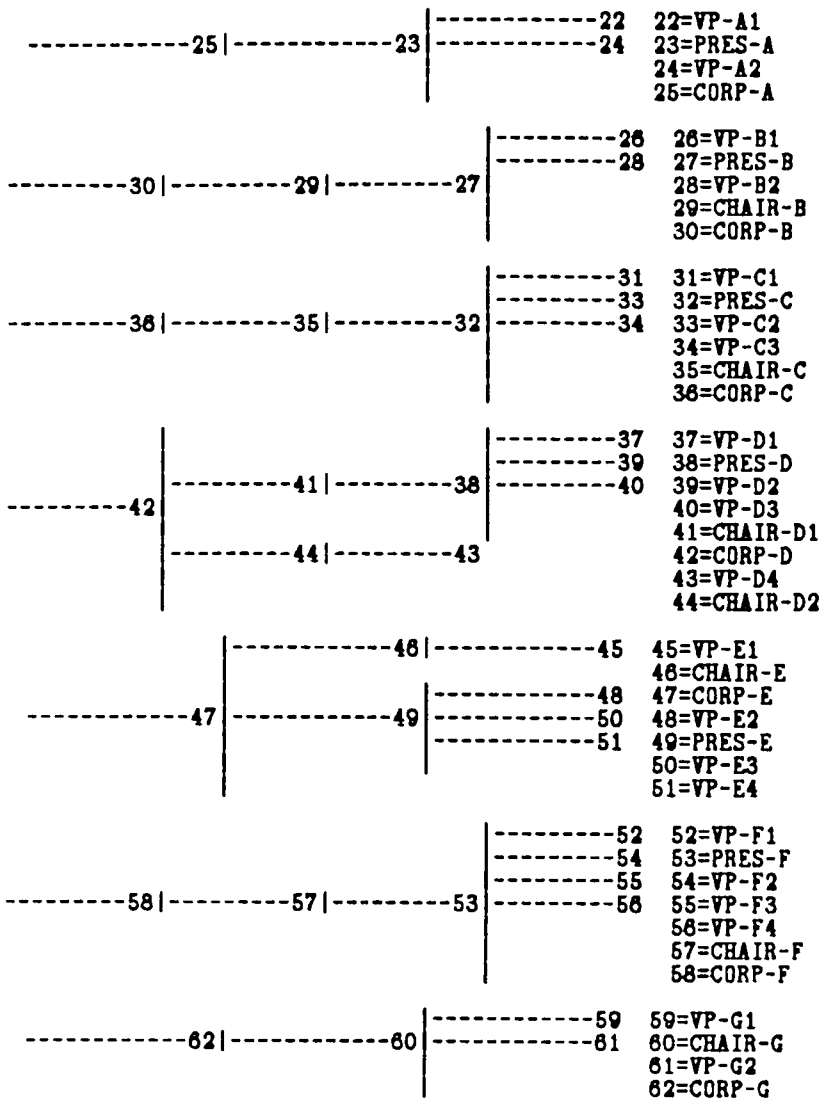
The president G-tree shows equivalent results. This is because each instance F-tree has most of its "structure" below the level of the president. By this we mean that the only difference between the top-level memette (CORP#) in the F-trees and the PRES# memette is that a couple of levels are interposed. Hence, any memettes above the level of the president must have a similar G-tree arrangement. This is due to the point raised in Chapter 5: "if an object varies at a low level of detail then it must also vary at all levels above that". What we have observed in this example is: the structure of the F-trees are crucial in determining the structure of the G-tree. This is to be expected from the integration of representation and generalization.

The remaining three presidents that appear indexed under the PRES# G-tree (&MEM28, &MEM29, and &MEM30) have no subordinates in their F-trees. Therefore, they are not similar to the other presidents, and get categorized as anomalies near the top of the G-tree. The vice-president G-tree shows that all the vice-presidents are categorized on the same level. Again, this is indicative of the fact that they have no F-tree structure below them (i.e., they have no F-children), and that no other means of distinguishing memettes have been used (i.e., relations or structure-independent data).

6.3.3 A sample run

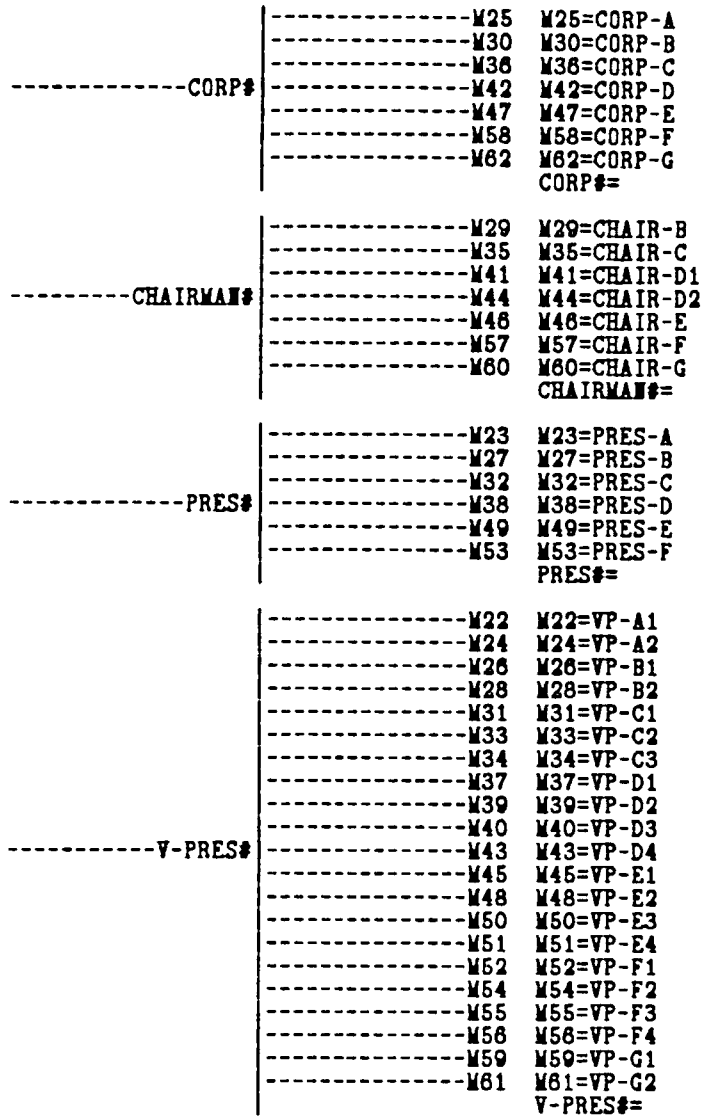
It is instructive to follow a run of CORPORATE-RESEARCHER through its incremental learning process. In the next ten figures a sample run is demonstrated using output from the system. Because F-trees of real corporations tend to be large, hypothetical corporations with very small F-trees will be used. Seven corporations (CORP-A through CORP-G) are included in this run, the largest of which has eight memettes. All the essential features of CORPORATE-RESEARCHER are demonstrated in this example.

Figure 6-14 shows the F-trees of all seven corporations. We have not used any function information nor relations in these representations in order to be able to clearly make our points. Only four initial memettes are required to establish a basis for comparison of the F-children in this example. These four memettes are shown in Figure 6-15 as the roots of four G-trees. Initially, each G-tree classifies memettes that are variants of the root memette. In this way, the initial G-trees serve as a cross-reference listing of the memettes comprising the F-trees. During the course of this program run, the G-trees will continually undergo structural changes resulting in a hierarchical categorization of the memettes that begin as single-level hierarchies.



The F-trees of seven corporations are shown. The names given to each memette indicate what they are variants of (CORP--corporation, CHAIR--chairman-of-the-board, PRES--president, and VP--vice-president).

Figure 6-14: Seven hypothetical corporations



These G-trees show how the memettes in the F-trees in Figure 6-14 are initially classified (i.e., what concepts the memettes are initially variants of). This classification is needed as a basis for comparing F-children during generalization.

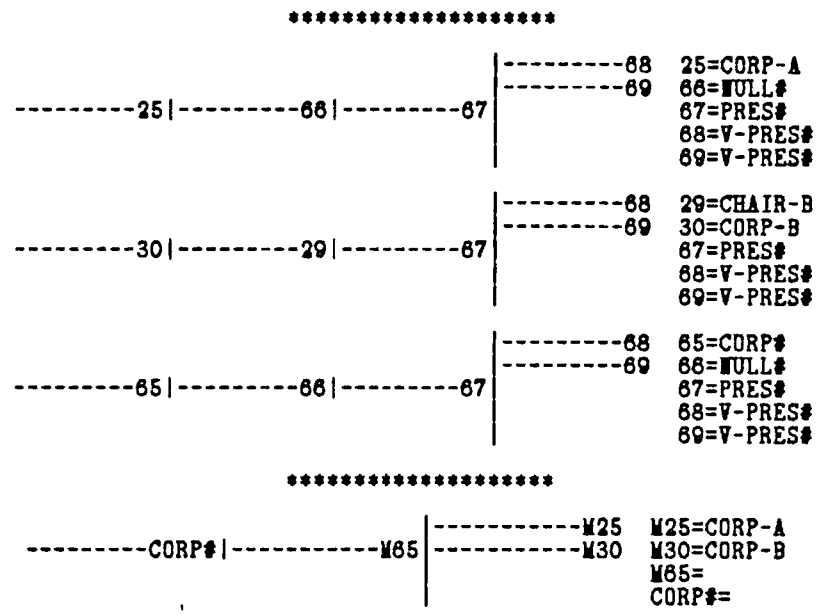
Figure 6-15: Initial configuration of four G-trees.

The run begins by adding CORP-A to the empty CORP# G-tree (not shown). This is done simply to create a starting point. Next, CORP-B is generalized into the knowledge base. This is a two-part process. First, CORP-B's F-tree is matched against that of CORP-A (since it is the only one in memory so far) to determine the F-children correspondence. Then, the results of this matching are used to incorporate CORP-B's F-tree into memory. As shown in Figure 6-16, a level-hop was required in order to get the best match of F-children. A null memette (NULL#) was inserted in between the top-level memette in CORP-A and the president, it matches to CHAIR-B. Consequently, the generalized concept of a corporation (&MEM65) created by this process also has a null memette where the

chairman would be. Notice that all F-children below this point have been inherited by both CORP-A and CORP-B, indicating the strong similarity between these two corporations. The reason that CORP-A and &MEM65 both appear, even though they are essentially identical, is that we purposely keep all top-level memettes in the knowledge base. This allows the CORP# G-tree to act as a true categorization device (i.e., it can be used to determine how similar or different corporations are, by inspection). The bottom diagram in Figure 6-16 shows how the CORP# G-tree (the one of most interest) appears so far.

CHAIR-B (&MEM29) was made a variant of NULL# (&MEM66) as a consequence of generalization. In order to keep its identity, the ALTERNATE-VARIANT-OF slot is needed. In this case, the ALTERNATE-VARIANT-OF slot in CHAIR-B was filled with CHAIR#, indicating that this memette is an instance of a chairman, despite the fact that it is structurally a VARIANT-OF NULL#.

```
*(GEN B)
Matching &MEM30 against &MEM25 .... 76
Best match is:
(76 ((&MEM25 . &MEM30)
     ((NULL# . &MEM29)
      ((&MEM23 . &MEM27) ((&MEM24 . &MEM28)) ((&MEM22 . &MEM26))))))
Incorporating into g-tree ...
New generalization created: &MEM65
with variants: (&MEM30 &MEM25)
```

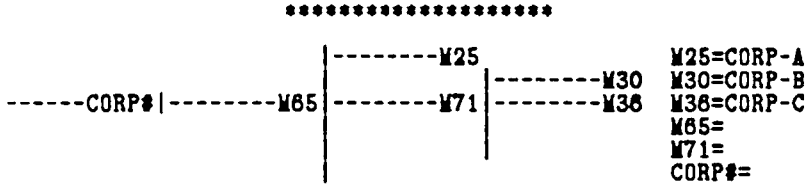


The top diagram shows the results of the matching process, after the GEN function is called. CORP-B in matched against CORP-A with a result of 76 points. The "best match" is a LISP structure that shows the memette frame correspondence found. Note that NULL# was inserted in order to achieve a best match. The middle diagram shows the resultant F-trees for CORP-A, CORP-B, and the newly created concept, &MEM65. At the bottom, the CORP# G-tree is shown, without including the other corporations yet to be generalized.

Figure 6-16: First of six incremental generalizations.

CORP-C is the next F-tree to be added into the unified memory structure. It is found to most closely match CORP-B (they are exactly the same, except CORP-C has an extra vice-president). A new concept must be created with their common F-children (&MEM71). Figure 6-17 shows the results of this match and the CORP# G-tree after generalizing. All three memettes in the G-tree, prior to forming the generalization, were tried as possible matches. The score passed back from the F-tree matching algorithm (described in Chapter 5) is given after each attempted match. The match with the highest score is chosen as the G-tree node at which to index the new F-tree.

```
*(GEN C)
Matching &MEM36 against &MEM65 .... 67
Matching &MEM36 against &MEM30 .... 103
Matching &MEM36 against &MEM25 .... 67
Best match is:
(103 ((&MEM30 . &MEM36)
      ((&MEM29 . &MEM35)
       ((&MEM67 . &MEM32) ((&MEM68 . &MEM34)) ((&MEM69 . &MEM33))))))
Incorporating into g-tree ...
New generalization created: &MEM71
with variants: (&MEM36 &MEM30)
```

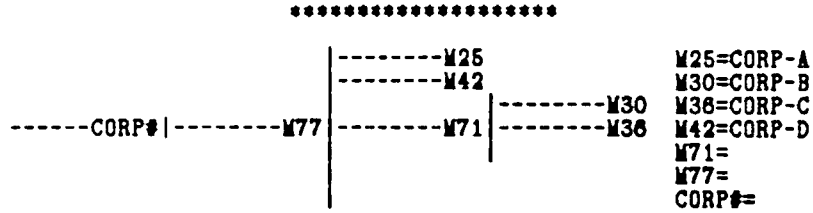


The top diagram shows the process of determining the "correct" place to locate CORP-C in the CORP# G-tree. The algorithm first compares CORP-C with the root of the G-tree, then to all of its variants. If any variant results in a better match, then that branch of the G-tree is followed. If there is a tie score, the root node is used.

Figure 6-17: CORP-C is added into memory.

Figure 6-18 shows similar output for incrementally generalizing CORP-D into memory. It was found to match most closely with a generalized concept, &MEM65 (which becomes &MEM77). Furthermore, there was no need to create a new concept memette because the intersection of the three F-trees, &MEM42, &MEM25, and &MEM71 is equivalent to the intersection of just &MEM25 and &MEM71. The reader may notice that a score of 52 was found in all three attempted matches. When this occurs, the program chooses the location that is highest in the G-tree as the place to index the new instance F-tree. There is no reason to make more specific generalizations than the data warrants.

```
*(GEN D)
Matching &MEM42 against &MEM65 .... 52
Matching &MEM42 against &MEM71 .... 52
Matching &MEM42 against &MEM25 .... 52
Best match is:
(52 ((&MEM65 . &MEM42)
     ((&MEM66 . &MEM41)
      ((&MEM67 . &MEM38) ((&MEM68 . &MEM40)) ((&MEM69 . &MEM39))))))
Incorporating into g-tree ...
Made &MEM42 a variant of &MEM77
```



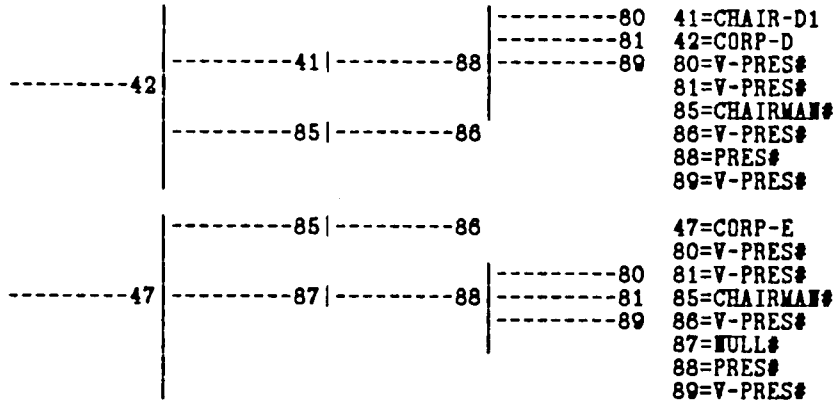
In this case there was a tie score, so the comparison of CORP-D against &MEM65 was used for incorporation into memory. (The text describes why.) Note that, although a new concept was not formed, &MEM65 was eliminated and &MEM77 took its place. This is an artifact of the program, and is not at all significant.

Figure 6-18: Incorporating CORP-D into the G-trees.

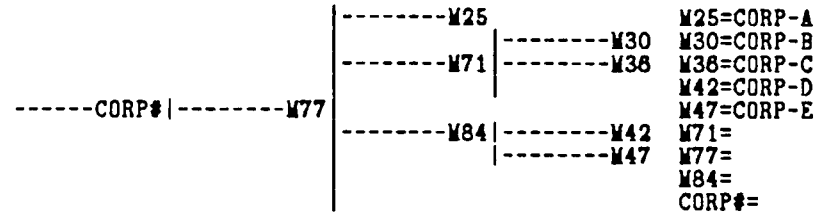
In the next cycle of the program, we can see that level-hopping was needed again, and that the scoring mechanism can be used to indicate the degree of similarity of two representations. The top diagram in Figure 6-19 illustrates that CORP-E (&MEM47) matches CORP-D (&MEM42) much better than any of the others that were tried (a score of 139 verses 39). Consequently it was generalized against it forming a new concept (&MEM84). Although these two corporations are virtually identical, it was necessary to insert a null memette between the top-level of CORP-E and its president in order to get the best result. (Although the level-hop caused the matching algorithm to find a better overall match (i.e., a higher score), the insertion of a NULL# memette actually decreases the score returned by the matcher. The negative scoring effect is intended to limit the use of level-hopping so that too many levels are not inserted. See Appendix A for the details of scoring level-hops.) This indicates that there may be a chairman missing from the representation of CORP-E, but there is not enough evidence in the data seen so far to assume it should be there.

```
*(GEN E)
Matching &MEM47 against &MEM77 .... 36
Matching &MEM47 against &MEM25 .... 36
Matching &MEM47 against &MEM71 .... 36
Matching &MEM47 against &MEM42 .... 139
Best match is:
(139 ((&MEM42 . &MEM47)
      ((&MEM44 . &MEM46) ((&MEM43 . &MEM45)))
      ((&MEM41 . NULL#)
       ((&MEM38 . &MEM49)
        ((&MEM37 . &MEM51))
        ((&MEM80 . &MEM50))
        ((&MEM81 . &MEM48))))))
Incorporating into g-tree ...
New generalization created: &MEM84
with variants: (&MEM47 &MEM42)
```

```
                    **********
```




```
                    **********
```

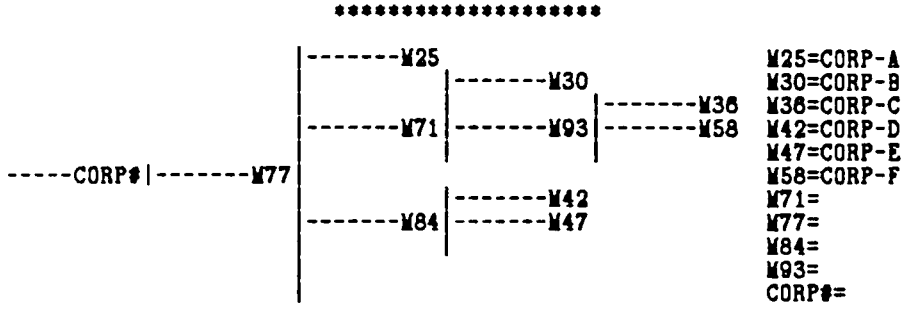


A very strong match was found between CORP-E and CORP-D (&MEM42). A level hop was needed in order to achieve such a high score, as is shown in the middle diagrams. The bottom diagram shows the CORP# G-tree is beginning to attain some structure.

Figure 6-19: The fourth corporation to be generalized into memory.

We can see the algorithm that locates the correct location in the G-tree at work in Figure 6-20. Here, CORP-F is being incorporated into memory, and is found to most closely match CORP-C (&MEM36). CORP-F is exactly the same as CORP-C, save for the addition of one vice-president.

```
*(GEN F)
Matching &MEM58 against &MEM77 .... 58
Matching &MEM58 against &MEM84 .... 58
Matching &MEM58 against &MEM25 .... 58
Matching &MEM58 against &MEM71 .... 66
Matching &MEM58 against &MEM36 .... 117
Matching &MEM58 against &MEM30 .... 94
Best match is:
(117 ((&MEM36 . &MEM58)
      ((&MEM35 . &MEM57)
       ((&MEM32 . &MEM53)
        ((&MEM31 . &MEM56))
        ((&MEM80 . &MEM55))
        ((&MEM81 . &MEM54))))))
Incorporating into g-tree ...
New generalization created: &MEM93
with variants: (&MEM58 &MEM36)
```



The location algorithm looks at 6 of the 8 members of the G-tree to find the best place for CORP-F. It is found at the deepest level. In general, if the G-tree is N levels deep with a branching factor of M the algorithm we currently use would look at a maximum of M*(N-1) locations.

Figure 6-20: CORP-F's incorporation into the knowledge base.

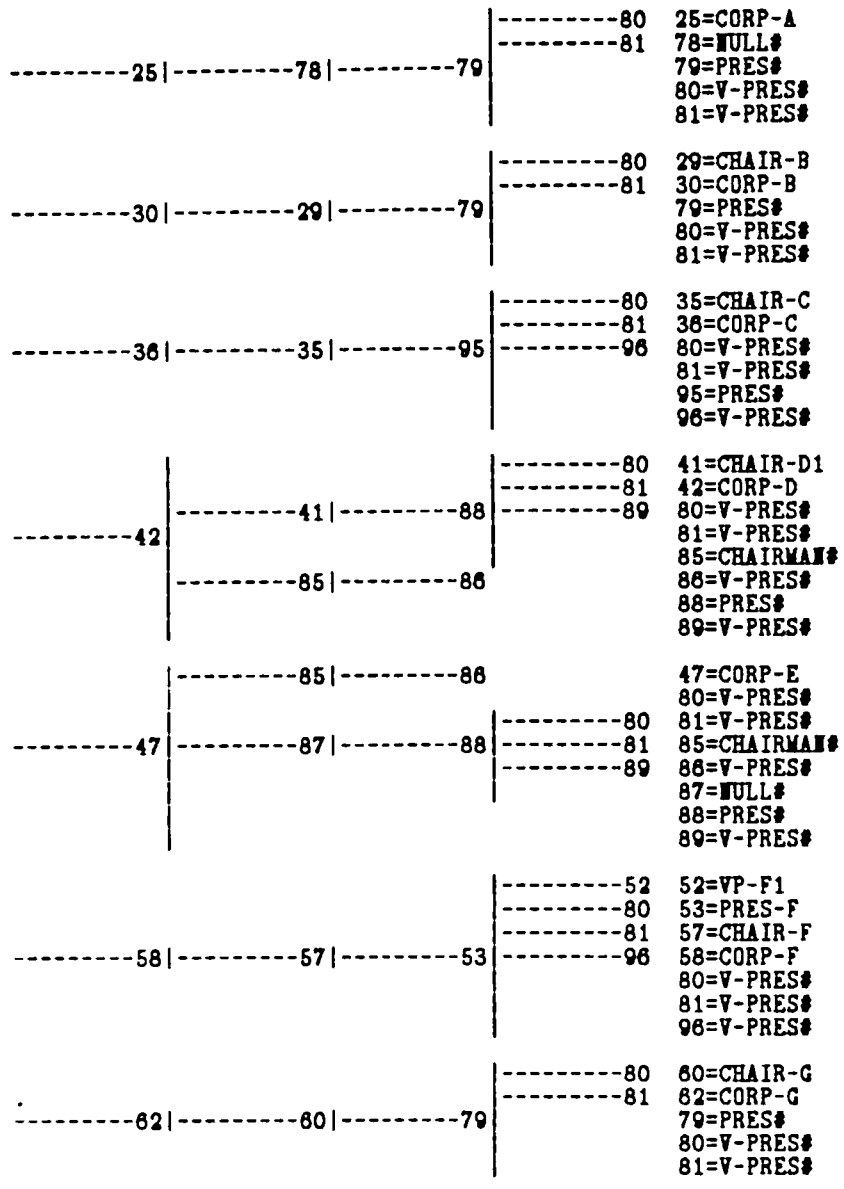
The last of the seven F-trees, CORP-G, is incorporated into the knowledge base as shown in Figure 6-21. Of particular interest, is the fact that the president, that was missing in the original F-Tree (see Figure 6-14), has been assumed (see Figure 6-22) This was made possible because of an increased confidence level in the generalization that: "all chairman have a president below them". Thus, level-hopping was used not just to get a better match grade, but also to allow for the incorporation of missing data.

```
*(GEN G)
Matching &MEM62 against &MEM77 .... 49
Matching &MEM62 against &MEM84 .... 49
Matching &MEM62 against &MEM25 .... 49
Matching &MEM62 against &MEM71 .... 57
Matching &MEM62 against &MEM93 .... 20
Matching &MEM62 against &MEM30 .... 85
Best match is:
(85 ((&MEM30 . &MEM62)
     ((&MEM29 . &MEM60)
      ((&MEM79 . NULL#) ((&MEM80 . &MEM61)) ((&MEM81 . &MEM59))))))
Incorporating into g-tree ...
New generalization created: &MEM101
with variants: (&MEM62 &MEM30)
```

CORP-G is brought into memory and found to match CORP-B (&MEM30) most closely. A level-hop is needed where the president should be. During the process of incorporation into the G-tree, the null memette from the level-hop was made a variant of PRES# (see Figure 6-22).

Figure 6-21: The final generalization of this run.

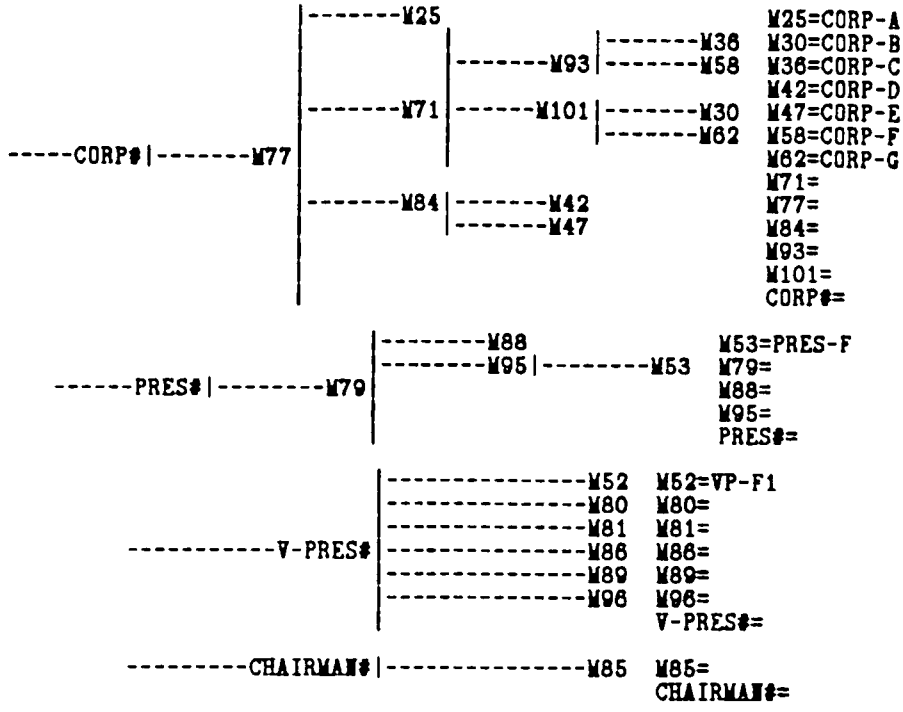
Figure 6-22 shows the seven corporate F-trees after they have all been incorporated into memory. Notice that the only structural changes that have been made occur where level-hopping was used. However, almost all of the original lower-level F-children have been eliminated and are now inherited from generalized concepts (not shown). In particular, all seven corporations inherit two of their vice-presidents from the same source (&MEM80 and &MEM81).



This diagram shows the F-trees of all seven corporations after the unified memory structure has been built. Any memettes with a number greater than 62 have been inherited from generalized concepts.

Figure 6-22: The final instance F-trees.

The final G-tree structures are shown in Figure 6-23. The CORP# G-tree has a great deal of structure that categorizes the input F-trees. It looks somewhat binary in nature, but this is just a consequence of the data it was fed and the order in which it was processed. The way it has categorized the instance corporations seems to correspond with how a person might do it.



This figure shows the G-trees after incorporation of all seven F-trees. It does not categorize most of the chairman, presidents, and vice-presidents because they are not kept in memory if they are not unique. The CORP# G-tree does categorize all corporations because CORPORATE-RESEARCHER keeps all top-level memettes in the instance F-trees.

Figure 6-23: The final G-tree structures.

The other G-trees (PRES#, V-PRES#, and CHAIRMAN#) show much less structure. This is because: 1- they categorize memettes with fewer F-children (e.g., vice-presidents have no F-children in this sample run), 2- CORPORATE-RESEARCHER does not keep memettes around if they do not need to be kept, except for the top-level F-tree memettes (which accounts for the bushy CORP# G-tree).

Intermediate nodes (i.e., not the root node or a leaf node) in a G-tree have reasonable (if not interesting) interpretations. In the CORP# G-tree, intermediate nodes (e.g., &ME93) represent generalized corporations that have the F-tree structure common to all its variants. (&MEM93 represents a corporation with a chairman, president, and three vice-presidents.) In the PRES# G-tree, &MEM95 represents a president that has three vice-presidents reporting to him. &MEM53 is a variant of &MEM95 because it has an additional vice-president in it F-tree. Thus, intermediate G-tree nodes are F-trees of generalized concepts that are less specific than instance F-trees and more specific than F-trees higher up in the G-tree.

6.3.4 Performance evaluation

From studying this sample run and several others involving much larger F-trees, we have found CORPORATE-RESEARCHER's performance to be as expected. Its behavior is predictable and in accordance with our design goals. Since the input to the program is tightly controlled by having to hand-code the instance F-trees, one would anticipate entirely predictable behavior. On occasion, it makes some generalizations that one would not expect by looking at the corporate charts -- some are good, others bad. The cause of these "bad" generalizations is due to the incorrect determination of the location in the G-tree where an F-tree should be incorporated. The algorithm currently in use is essentially a "greedy" method and can be misled. If all nodes in the G-tree were considered, CORPORATE-RESEARCHER would perform better in some cases, but would become computationally infeasible.

The program is memory efficient, because it removes unneeded memettes and makes heavy use of inheritance. However, it is not very time efficient. It spends most of its time matching one F-tree against another. In order to achieve level-hopping at any level, and possibly at several levels simultaneous, CORPORATE-RESEARCHER tries to insert null memettes above and below each real memette in both F-trees. The process of matching two 4-level, single-lineage F-trees requires 36 comparisons of trees. There may be a faster algorithm to perform this type of matching, but one solution may be to simply limit the amount of level-hopping permitted.

CORPORATE-RESEARCHER's limited types of data to generalize about cause it to produce predictable generalizations. However, this constraint also makes it a good program with which to demonstrate the principles of MERGE.

6.4 RESEARCHER

RESEARCHER is a large natural language processing system designed to read and understand patent abstracts. It uses the MERGE scheme to achieve this "understanding", but much of the program is concerned with parsing the textual input into F-tree representations.

The purpose of the RESEARCHER system is to form an intelligent database for a particular class of complex physical objects. At present, RESEARCHER's primary domain is computer disc drives. The database is intended to automatically categorize instances directly from the input text (i.e., without human intervention, as was needed in CORPORATE-RESEARCHER). It is also used as the knowledge source for an intelligent question answering system [Paris 84].

The program has the potential to be useful for patent searching by indexing patents according to their content. Current computerized patent searching systems

use a keyword-based retrieval which is not intelligent. RESEARCHER can also serve as an engineering knowledge system about how physical objects in one domain differ from others in the same domain. It could also be used in conjunction with a question answering module designed to instruct users on how disc drives are constructed and how they work.

One of the main difficulties in RESEARCHER is the correct parsing of the input text into an F-tree. Although the complex physical objects described by patent abstracts are hierarchical in nature (i.e., objects are described as parts with subparts, etc), the English language descriptions of them may not be tree-like in structure, and are usually incomplete. Thus, another use of RESEARCHER's memory is to help in processing future input (see [Lebowitz 84]).

The basic difference between the use of MERGE in RESEARCHER and in CORPORATE-RESEARCHER is that CORPORATE-RESEARCHER was built to demonstrate how a MERGE-based system operates, while RESEARCHER was a large system in need of some way to organize its knowledge intelligently. Consequently, RESEARCHER and the MERGE scheme have been developed simultaneously. This section is included in the thesis to show a real-world use for a MERGE-based understander system -- not to demonstrate how MERGE works in detail, as was done in the previous section.

6.4.1 RESEARCHER's domain

Patent abstracts are short summaries of full patents issued by the U. S. Patent Office. The abstracts that RESEARCHER reads are about physical objects. They account for a large percentage of all the patents issued. (Others types include: process patents, design patents, agriculture patents, and chemical patents.) They are difficult to read, mostly because they are written in a form of legalese. As a consequence, it is a painful experience for a layman to read through many of them trying to determine how to generalize about what they are describing. Thus, RESEARCHER, being a run on a machine, has the potential to perform better than humans do when processing many patents.

The basic premise behind using a MERGE-based understanding scheme in RESEARCHER is that complex physical objects are often hierarchically structured. That is, the PART-OF F-rel can be used to represent that one component is included within another. Of course, physical objects are additionally described in terms of the relations among their components. Thus, RESEARCHER makes heavy use of relations, using the representation scheme described in Section 4.5.

Although patent abstracts are precisely worded (for the most part), their descriptions are often incomplete. Because they are abstracts, they do not necessarily describe all the components that comprise an entire object. The focus of the text tends to be on the patentable part of a particular object. For example,

a patent abstract for an ultra-high speed disc drive would most likely describe the aspects of the invention that allow it to operate at a high speed -- not how the disc drive is constructed in its entirety. Consequently, RESEARCHER needs the ability to level-hop, and to assume missing data when necessary, as MERGE provides.

In this section, some sample patent abstracts will be presented in order to show how an augmented F-tree representation of them is obtained. Following this, a short run of the program will be shown in which aspects of MERGE that are particularly important in RESEARCHER will be emphasized.

6.4.2 Patent abstracts and F-trees

RESEARCHER parses patent abstracts into F-tree representations that are subsequently incorporated into its unified memory structure. The parsing process is driven by reading each word in the input, looking it up in a dictionary, applying its definition, and constructing whatever memettes, relations, and modifiers (described below) are needed for the correct representation (see [Lebowitz 80; Birnbaum and Selfridge 81] for descriptions of RESEARCHER-like text processing). The result of this processing is one or more augmented F-trees.

In Figure 6-24, a typical patent abstract for a disc drive is shown along with the output produced while parsing the first dozen words. (This abstract is taken from a real U S patent.) It is largely self-explanatory, but a few notes are in order. RESEARCHER allows a phrasal lexicon. This permits phrases like "at least" to be processed without having to understand the words separately. An *MP* is a memory pointer word that points to a memette frame.

In addition to physical relations among objects, functional or purpose relations are also processed. (Purpose relations are distinguished by the "P-" prefix, while physical relations have the "R-" prefix.) Purpose relations work just like physical relations (i.e., they have a characteristic and arguments), but there is, as yet, no underlying canonical scheme for classifying them. RESEARCHER also processes properties of memettes. This memette slot is used to capture a diverse range of features including: quantity, size, shape, color, etc. Most modifiers (usually adjectives) get placed into this slot. (There are other word classes that are processed but not shown in these examples.)

```
Patent: P7

(A DISC DRIVE INCLUDING AT LEAST ONE DISC *COMMA* MEANS FOR MOUNTING SAID
DISC *SEMI* MEANS FOR DRIVING SAID MOUNTING MEANS TO ROTATE THE DISC *COLON*
AT LEAST ONE TRANSDUCER COOPERATING WITH THE SURFACE OF SAID DISC TO READ
AND WRITE INFORMATION ON THE SURFACE *COMMA* MOUNTING MEANS FOR MOUNTING
SAID TRANSDUCER FOR COOPERATION WITH THE DISC *SEMI* A CARRIAGE FOR SAID
TRANSDUCER MOUNTING MEANS *COLON* THREE SPACED BEARINGS HAVING GROOVED OUTER
SURFACES MOUNTED ON SAID CARRIAGE *SEMI* A FIXED CYLINDRICAL TRACK ADAPTED
TO RECEIVE TWO OF SAID BEARINGS TO GUIDE THE CARRIAGE *SEMI* A SPRING-LOADED
CYLINDRICAL TRACK ADAPTED TO ENGAGE THE OTHER BEARING AND URGE SAID TWO
BEARINGS AGAINST THE FIXED TRACKS WHEREBY THE BEARINGS ARE CENTERED ON SAID
TRACKS FOR MOVEMENT THEREALONG AND MEANS FOR DRIVING THE CARRIAGE TO MOVE
THE CARRIAGE ALONG SAID TRACKS SO THAT THE TRANSDUCERS ARE MOVED RADIALLY
ALONG THE DISC SURFACE *STOP*)

Processing:

A               : New instance word -- skip
DISC DRIVE      : Phrase
-> DISC-DRIVE   : MP word -- memette DISC-DRIVE#
New DISC-DRIVE# instance (&MEM15)
INCLUDING       : Parts of &MEM15 (DISC-DRIVE#) to follow
AT LEAST        : Phrase
-> AT-LEAST     : Modifier modifier -- save and skip
ONE             : Memette modifier; save and skip
DISC            : MP word -- memette DISC#
New DISC# instance (&MEM16)
Augmenting &MEM16 (DISC#) with feature: NUMBER = GE
Assuming &MEM16 (DISC#) is part of &MEM15 (DISC-DRIVE#)
*COMMA* (*COMMA*1)
                : Break word -- skip
MEANS           : MP word -- memette UNKNOWN-THING#
New UNKNOWN-THING# instance (&MEM17)
Assuming &MEM17 (UNKNOWN-THING# -- 'MEANS') is part of &MEM15 (DISC-DRIVE#)
FOR (FOR1)      : Purpose indicator -- skip
MOUNTING        : Purpose word -- save and skip
SAID            : Antecedent word -- skip
DISC            : MP word -- memette DISC#
Reference for DISC#: &MEM16
Establishing P-SUPPORTS relation; SUBJECT: &MEM17 (UNKNOWN-THING# --
'MEANS'); OBJECT: &MEM16 (DISC#) [&REL1]
*SEMI*          : Break word -- skip
  .
  .
  .
```

The text of a disc drive patent is shown at the top of this figure. The first several words that RESEARCHER has processed are shown below it, along with the running comments that the program outputs

Figure 6-24: Parsing a patent

The focus of MERGE is primarily on F-trees augmented by relations, so we will continue to only explain this aspect of the sample output from RESEARCHER. Figure 6-25 shows the augmented F-tree produced from a complete parse of Patent P7. The top diagram and list of relations is syntactically the same as in CORPORATE-RESEARCHER. The large number of relations indicated along the F-rel links is due to the fact that patents describe many more relations than do corporate charts, and that purpose relations are included in this diagram. Only three of the 23 relations (letters A through W) are shown below the F-tree in Figure 6-25. The column to the right of the tree shows the text name of each memette along with a left-truncated list of memette modifiers. For example memette &MEM24 represents "three spaced bearings", and is shown in detail (i.e., the memette frame with all its slots) at the bottom of the figure.

```
Text Representation:

                |----ACEFJ-16|----DGHVW-20                15=DISC-DRIVE
                |                                         16=DISC
                |------ABC-17                             17=MEANS
                |--------B-18                             18=MEANS
                |----DIJVW-19                             19=TRANSDUCER
                |------EFG-21                             20=SURFACE
                |-------HI-22|-----LSTU-23|----------25    21=INFORMATION
----------15    |                                         22=MEANS
                |                |----------25            23=CARRIAGE
                |----------24    |----KLMOP-27            24=SEPARATE BEARINGS
                |                |-------MN-29            25=EXTERIOR SURFACES
                |--------K-26                             26=CYLINDRICAL TRACK
                |---MOPQRU-28                             27=2 1 BEARINGS
                |------QRS-30                             28=TRACKS TRACK
                |                                         29=1 1 BEARING
                |                                         30=MEANS

A list of relations:

           Subject:                Relation:          Object:

[&REL1/A]  &MEM17 ('MEANS')        {P-SUPPORTS}       &MEM16 (DISC#)

[&REL4/D]  &MEM19 (TRANSDUCER#)    {R-ADJACENT-TO}    &MEM20 (SURFACE#)

[&REL7/G]  &MEM21 (DATA#)          {R-ON-TOP-OF}      &MEM20 (SURFACE#)

    .
    .
    .

                          **********

     Memette Frame: &MEM24

     REC-TYPE        = &MEM
     ID              = ID46971
     VARIANT-OF      = BEARING#
     CLASS           = NOUN
     TYPE            = COMPOSITE
     TEXT-NAME       = (BEARINGS)
     PROPERTIES      = ((NUMBER 3) (DISTANCE SEPARATE))
     COMPONENTS      = (&MEM29 &MEM27 &MEM25)
```

The top part of this diagram shows the F-tree, built by RESEARCHER, for Patent P7. A partial list of relation descriptions is given below that. The bottom figure is an exploded view of what a memette in the program contains. Of particular interest are the PROPERTIES slot, the TEXT-NAME slot, and the COMPONENTS slot.

Figure 6-25: The F-tree for Patent P7.

Figure 6-26 shows the text and F-tree representation of another patent, P32 (slightly modified from the original to accentuate the generalization process). We have not presented the list of relations nor any of the output from the parsing process. This F-tree will be generalized with the one for Patent P7.

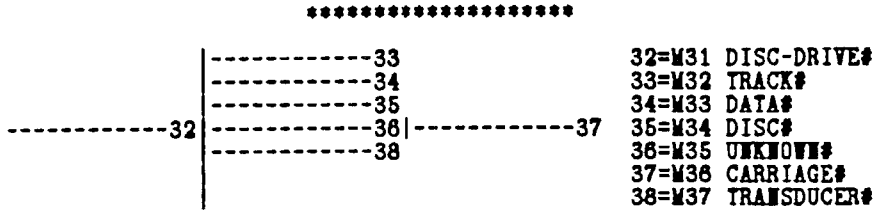
```
Patent: P32

(A FLOPPY DISK MAGNETIC INFORMATION STORAGE AND RETRIEVAL UNIT HAVING A
MAGNETIC READ-WRITE HEAD *COMMA* A CARRIAGE AND APPARATUS FOR SHIFTING THE
CARRIAGE RADIALLY WITH RESPECT TO THE FLOPPY DISK TO EFFECT DATA TRANSFER
WITH SELECTABLE TRACKS ON THE DISK *PERIOD* THE CARRIAGE SHIFTING APPARATUS
INCLUDES A LEAD SCREW SHAFT *SEMI* A CARRIAGE NUT ARRANGED ON THE SHAFT AND
MOVEABLE AXIALLY WHEN THE SHAFT IS ROTATED *SEMI* AND A STEPPER MOTOR HAVING
A ROTOR ARRANGED COAXIALLY WITH *COMMA* AND RIGIDLY ATTACHED TO ONE END OF
THE SHAFT *PERIOD* TWO BEARINGS ARE PROVIDED AS PART OF THE DISC DRIVE TO
HOLD THE SHAFT AND ROTOR COMBINATION AT OPPOSITE ENDS OF THE SHAFT *STOP*)

Text Representation:

                |------CE-1                                  1=SOFT DISK
                |-----ACD-3                                  2=DISC-DRIVE
                |---------4                                  3=INFORMATION
                |          |------BF-6                       4=READ-WRITE-HEAD
                |          |          |------J-12            5=APPARATUS
                |          |-----GHI-8|------M-14|------K-13 6=CARRIAGE
-----AKL-2------|-------B-5|          |                      7=SOME TRACKS
                |          |------FG-9                       8=SCREW-SHAFT
                |          |--------10|----JLM-11            9=NUT
                |          |          |------J-12           10=SOME MOTOR
                |          |                               11=ROTOR
                |------DE-7                                12=1 END
                |                                          13=2 BEARINGS
                |                                          14=FACETS ENDS
```

Figure 6-26: Another disc drive patent and its F-tree representation.

The concept created from the generalization of Patent P7 and Patent P32 is shown in Figure 6-27. The top part of the figure displays the output from the generalizer, which, again, is syntactically the same as CORPORATE-RESEARCHER's. Memettes have been paired-up according to what they are variants of and how well their F-children match. (Relations also figure into this process, but are not shown here. An example of relation generalization is shown below.) An UNKNOWN# (&MEM36) memette appears in this generalization. It indicates that two memettes have been matched together because their F-children match. The variant memettes (&MEM5 and &MEM22) themselves are not similar -- they were VARIANTS-OF different initial concepts. The use of the UNKNOWN# memette (as opposed to the NULL# memette) is discussed in more detail below.

The generalized concept of a disc drive that has been created by using just these two patents is represented by the F-tree rooted at &MEM32. It represents a disc drive with a disc, data, a transducer (read/write head), a track, and an unknown assembly. This unknown assembly has a carriage as a component.

```
(GEN '&MEM15]
Matching &MEM15 against &MEM2  .... 228
Best match is:
(228 ((&MEM2 . &MEM15)
      ((&MEM7 . &MEM28))
      ((&MEM3 . &MEM21))
      ((&MEM1 . &MEM16))
      ((&MEM5 . &MEM22) ((&MEM6 . &MEM23)))
      ((&MEM4 . &MEM19))))
Incorporating into g-tree ...
New generalization created: &MEM32
with variants: (&MEM15 &MEM2)
```



The top diagram shows the process of generalization, the matching, and the incorporation into memory. The bottom figure shows the resultant generalized concept of a disc drive.

Figure 6-27: The generalization of Patents P7 and P32.

6.4.3 A sample run

As in the previous section, we will examine a sample run of RESEARCHER here. We use only four abstracts to illustrate how the program works, because much of its functioning is identical to CORPORATE-RESEARCHER's.

Instead of using real patents, textual descriptions of disc drives that are patent-like in language will be used. This method allows for a more focused discussion of the issues, as well as providing examples that will be correctly parsed. The chances of finding a set of four patent abstracts that are similar enough for our illustrations is extremely small. Therefore, we have written the four examples about to be presented.

In the following figures, only the output from RESEARCHER that is pertinent to the discussion will be shown. The basic cycle is: parse - generalize - parse. Thus, this sample run differs from that of CORPORATE-RESEARCHER in that it is more like the real world; one does not usually know all the data to be generalized about in advance, but rather, incrementally.

Figure 6-28 shows the text and its F-tree representation for sample Patent T1 (&MEM1). The three relations involved in the F-tree are shown at the bottom of the diagram.

```
Patent: T1

(A DISC DRIVE COMPRISING AN ENCLOSURE SURROUNDING THE DISC DRIVE *COMMA*
SAID DISC DRIVE INCLUDES A SPINNING ASSEMBLY A DISC AND A READWRITE HEAD
*COMMA* SAID SPINNING ASSEMBLY INCLUDES A SPINDLE CONNECTED TO A MOTOR
*COMMA* SAID ENCLOSURE COMPRISING A COVER ON TOP OF A SUPPORT MEMBER)

  Text Representation:

                    |                  |----------C-8   1=DISC-DRIVE
                    |----------A-2     |----------C-9   2=ENCLOSURE
                    |                  |                3=ROTATION ASSEMBLY
                    |                                   4=DISC
  ----------A-1     |                  |----------B-6   5=WRITE-HEAD
                    |-------------3    |----------B-7   6=SPINDLE
                    |                  |                7=MOTOR
                    |-------------4                     8=COVER
                    |-------------5                     9=MEMBER

A list of relations:

            Subject:                Relation:          Object:

[&REL1/A]  &MEM1 (DISC-DRIVE#)    {R-SURROUNDED-BY}  &MEM2 (ENCLOSURE#)
[&REL2/B]  &MEM6 (DRIVE-SHAFT#)   {R-CONNECTED-TO}   &MEM7 (MOTOR#)
[&REL3/C]  &MEM8 (COVER#)         {R-ON-TOP-OF}      &MEM9 ('MEMBER')
```

The text for a hypothetical patent abstract (Patent T1) is shown along this its F-tree, after being parsed. The bottom of this figure shows a list of the relations used in the augmented F-tree.

Figure 6-28: The first of four sample abstracts.

When Patent T2 (&MEM10) is parsed and incorporated into the unified memory structure, the need for level-hopping is apparent. The top of Figure 6-29 shows that Patent T2 is just like Patent T1 except that the DISC is part of a MAGNETIC ASSEMBLY. Patent T1 has the DISC directly as part of the DISC-DRIVE. If the discs in both patents are to be matched, there must be some method of dealing with it. Level-hopping is the solution. The results of the GEN function are shown in the middle diagram. A NULL# memette was inserted in order to achieve a good match. &MEM22 is the concept formed by the generalization of Patent T1 and Patent T2, it shows that NULL# has been included in the F-tree.

```
Patent: T2

(A DISC DRIVE COMPRISING AN ENCLOSURE SURROUNDING THE DISC DRIVE *COMMA*
SAID DISC DRIVE INCLUDES A SPINNING ASSEMBLY A MAGNETIC ASSEMBLY AND A
READ/WRITE HEAD *COMMA* SAID SPINNING ASSEMBLY INCLUDES A SPINDLE CONNECTED
TO A MOTOR *COMMA* SAID MAGNETIC ASSEMBLY COMPRISING A DISC *COMMA* SAID
ENCLOSURE COMPRISING A COVER ON TOP OF A SUPPORT MEMBER)

  Text Representation:

                    |                    |----------F-18  10=DISC-DRIVE
                    |----------D-11      |----------F-19  11=ENCLOSURE
                    |                    |                12=ROTATION ASSEMBLY
                    |                                     13=MAGNETISM ASSEMBLY
  ----------D-10    |                    |----------E-15  14=WRITE-HEAD
                    |------------12      |----------E-16  15=SPINDLE
                    |                    |                16=MOTOR
                    |------------13      |------------17  17=DISC
                    |------------14                       18=COVER
                    |                                     19=MEMBER

                                  **********

(GEN '&MEM10]
Matching &MEM10 against &MEM1 .... 170
Best match is:
(170 ((&MEM1 . &MEM10)
      ((&MEM5 . &MEM14))
      ((NULL# . &MEM13) ((&MEM4 . &MEM17)))
      ((&MEM3 . &MEM12) ((&MEM7 . &MEM16)) ((&MEM6 . &MEM15)))
      ((&MEM2 . &MEM11) ((&MEM9 . &MEM19)) ((&MEM8 . &MEM18)))))
Incorporating into g-tree ...
New generalization created: &MEM22
with variants: (&MEM10 &MEM1)

                                  **********

                    |------------23                       22=DISC-DRIVE#
                    |------------24      |------------25  23=TRANSDUCER#
                    |                    |                24=NULL#
                    |                    |----------H-27  25=DISC#
  ----------I-22    |------------26      |----------H-28  26=UNKNOWN-ASSEMBLY#
                    |                                     27=MOTOR#
                    |----------I-29      |----------G-30  28=DRIVE-SHAFT#
                    |                    |----------G-31  29=ENCLOSURE#
                    |                                     30=UNKNOWN-THING#
                    |                                     31=COVER#
```

The top diagram shows the text and F-tree for a patent abstract that is similar to Patent T1 but with an extra level added above the DISC. A NULL# memette is inserted during generalization, shown in the middle figure. The bottom diagram shows the generalized F-tree.

Figure 6-29: Level-hopping in RESEARCHER.

Relations play a much more important role in RESEARCHER than in CORPORATE-RESEARCHER. Therefore, we examine three types of simple relation generalization in the next example, shown in Figure 6-30. Patent T3 differs from Patent T1 (which it ultimately gets matched against) in that: 1- the word *encircling* is used instead of *surrounding*, 2- an *axle* is used in place of a *spindle*, 3- the *cover* is ABOVE the *support member* -- not ON-TOP-OF it.

In the first case, the words are defined in terms of the same relation, so that *encircling* and *surrounding* both use the R-SURROUNDED-BY relation definition. This leads to a canonical representation that lends itself to generalization. *Spindle* and *axle* have different meanings, so, in the next case, the memettes have no

common generalization and they are not included in the concept F-tree, &MEM42. Therefore, the relation generalization can not include a memette in the subject slot. The third type of relation generalization illustrated is that the arguments of the relations are the same but the characteristics don't compare. Because ABOVE and ON-TOP-OF are not the same, their generalization is NIL. Of course, they have something in common (i.e., they both indicate a vertical direction) but the matching criteria that we are using in this example is that they must match completely.

```
Patent: T3

(A DISC DRIVE COMPRISING AN ENCLOSURE ENCIRCLING THE DISC DRIVE *COMMA* SAID
DISC DRIVE INCLUDES A SPINNING ASSEMBLY A DISC AND A READWRITE HEAD *COMMA*
SAID SPINNING ASSEMBLY INCLUDES AN AXLE CONNECTED TO A MOTOR *COMMA* SAID
ENCLOSURE COMPRISING A COVER ABOVE A SUPPORT MEMBER)

  Text Representation:

                    |                    |----------L-39  32=DISC-DRIVE
                    |----------J-33      |----------L-40  33=ENCLOSURE
                    |                    |                34=ROTATION ASSEMBLY
                    |                                     35=DISC
  ----------J-32    |                    |----------K-37  36=WRITE-HEAD
                    |------------34      |----------K-38  37=AXLE
                    |                    |                38=MOTOR
                    |------------35                       39=COVER
                    |------------36                       40=MEMBER

A list of relations:

            Subject:                Relation:          Object:

[&REL10/J] &MEM32 (DISC-DRIVE#)  {R-SURROUNDED-BY}  &MEM33 (ENCLOSURE#)
[&REL11/K] &MEM37 (AXLE#)        {R-CONNECTED-TO}   &MEM38 (MOTOR#)
[&REL12/L] &MEM39 (COVER#)       {R-ABOVE}          &MEM40 ('MEMBER')

                              **********

(GEN '&MEM32]
Matching &MEM32 against &MEM22 .... 116
Matching &MEM32 against &MEM10 .... 102
Matching &MEM32 against &MEM1 .... 118
Best match is:
(118 ((&MEM1 . &MEM32)
      ((&MEM23 . &MEM36))
      ((&MEM26 . &MEM34) ((&MEM27 . &MEM38)))
      ((&MEM29 . &MEM33) ((&MEM30 . &MEM40)) ((&MEM31 . &MEM39)))))
Incorporating into g-tree ...
New generalization created: &MEM42
with variants: (&MEM32 &MEM1)

                              **********

                    |------------23                     23=TRANSDUCER#
                    |------------24|------------25      24=NULL#
                    |                                   25=DISC#
                    |                                   27=MOTOR#
  ----------0-42    |------------43|----------M-27      29=ENCLOSURE#
                    |                                   30=UNKNOWN-THING#
                    |----------0-29|----------M-30      31=COVER#
                    |              |----------M-31      42=DISC-DRIVE#
                    |                                   43=UNKNOWN-ASSEMBLY#

A list of relations:

            Subject:                Relation:          Object:

[&REL13/M] &MEM31                {NIL}              &MEM30
[&REL14/M]                       {R-CONNECTED-TO}   &MEM27
[&REL15/O] &MEM42                {R-SURROUNDED-BY}  &MEM29
```

Patent T3 differs from Patent T1 mainly in the way it uses relations. Of particular importance is the list of relations shown in the bottom diagram. Notice that some of the data is missing in these generalized relations. The text explains why.

Figure 6-30: Relation generalization.

The fourth and final patent will demonstrate how two different objects can be matched together because they have the same parts. This process may need to

occur when generalizing about real patent descriptions for several reasons including: parts are named differently by different inventors, one or both of the definitions is missing from RESEARCHER's vocabulary, some text was parsed incorrectly, and so forth.

In Figure 6-31 Patent T4 is shown to be identical to Patent T1 with one minor exception. A WIDGET is used to replace the ENCLOSURE in Patent T1. During the generalization process, the WIDGET is found to correspond to the ENCLOSURE because they both have the same parts (the COVER and the MEMBER). Thus, the generalized concept (&MEM59) uses an UNKNOWN# memette (&MEM64). Notice that the variant F-trees (&MEM22 and &MEM49) have not been altered by this generalization.

An UNKNOWN# memette is one that gets created as the result of generalizing two memettes that are ultimately variants of different concepts. They have been matched up because their F-tree structures are similar and/or because of other information that matches closely (e.g., relations, properties, etc.). Since they have no common ancestor in a G-tree, there is no common name that can be given to their generalization. Hence, their generalized concept becomes a variant of (and gets its name from) the UNKNOWN# memette.

An UNKNOWN# memette signifies that its variants are real memettes but they were not initially variants of the same concepts. The presence of a NULL# memette means that at least one of its variants was not included in the initial F-tree used to make the generalization, and therefore may or may not actually exist.

```
Patent: T4

(A DISC DRIVE COMPRISING A WIDGET SURROUNDING THE DISC DRIVE *COMMA* SAID
DISC DRIVE INCLUDES A SPINNING ASSEMBLY A DISC AND A READWRITE HEAD *COMMA*
SAID SPINNING ASSEMBLY INCLUDES A SPINDLE CONNECTED TO A MOTOR *COMMA* SAID
WIDGET COMPRISING A COVER ON TOP OF A SUPPORT MEMBER)

  Text Representation:

                    |                    |----------U-58  49=DISC-DRIVE
                    |----------S-50      |----------U-57  50=WIDGET
                    |                    |                51=ROTATION ASSEMBLY
                    |                                     52=DISC
  ----------S-49    |                    |----------T-54  53=WRITE-HEAD
                    |------------51      |----------T-55  54=SPINDLE
                    |                    |                55=MOTOR
                    |------------52                       56=COVER
                    |------------53                       57=MEMBER

                                  **********
(GEN '&MEM49]
Matching &MEM49 against &MEM22 .... 121
Matching &MEM49 against &MEM42 .... 121
Matching &MEM49 against &MEM10 .... 84
Best match is:
(121 ((&MEM22 . &MEM49)
      ((&MEM23 . &MEM53))
      ((&MEM26 . &MEM51) ((&MEM27 . &MEM55)) ((&MEM28 . &MEM54)))
      ((&MEM29 . &MEM50) ((&MEM30 . &MEM57)) ((&MEM31 . &MEM56)))))
Incorporating into g-tree ...
New generalization created: &MEM59
with variants: (&MEM49 &MEM22)
                                  **********

                    |------------24|------------25  22=DISC-DRIVE#
                    |                               24=NULL#
                    |                    |----------R-65  25=DISC#
                    |----------P-29      |----------R-66  29=ENCLOSURE#
  ----------P-22    |                    |                60=TRANSDUCER#
                    |------------60                       61=UNKNOWN-ASSEMBLY#
                    |------------61      |----------Q-62  62=MOTOR#
                    |                    |----------Q-63  63=DRIVE-SHAFT#
                    |                                     65=UNKNOWN-THING#
                    |                                     66=COVER#

                    |                    |----------U-65  49=DISC-DRIVE
                    |----------S-50      |----------U-66  50=WIDGET
                    |                    |                52=DISC
                    |                                     60=TRANSDUCER#
  ----------S-49    |------------52                       61=UNKNOWN-ASSEMBLY#
                    |------------60                       62=MOTOR#
                    |------------61      |----------T-62  63=DRIVE-SHAFT#
                    |                    |----------T-63  65=UNKNOWN-THING#
                    |                                     66=COVER#

                    |------------60                       59=DISC-DRIVE#
                    |                    |----------W-62  60=TRANSDUCER#
                    |------------61      |----------W-63  61=UNKNOWN-ASSEMBLY#
  ------------59    |                                     62=MOTOR#
                    |------------64      |----------V-65  63=DRIVE-SHAFT#
                    |                    |----------V-66  64=UNKNOWN#
                    |                                     65=UNKNOWN-THING#
                    |                                     66=COVER#
```

The top diagram shows Patent T4 which uses a WIDGET in place of the ENCLOSURE. However, it still can match to the ENCLOSURE by virtue of the fact that its components are identical. The bottom figure shows the variant F-trees as well as the newly created generalization.

Figure 6-31: Matching dissimilar memettes.

6.4.4 Performance evaluation

RESEARCHER performs adequately when it is given simple text to process. The four examples shown above were easy to come up with and clearly demonstrate the workings of the program. Level-hopping works, as do the basic operations of inheritance, addition, subtraction, and substitution. Relation generalizing works, and is being extended to make comparisons of characteristics at the primitive level. This allows RESEARCHER to capture the similarity of ON-TOP-OF and ABOVE, for example.

There are some technical problems with RESEARCHER's text processing. However, the real difficulty lies in the fact that the abstracts are not describing the same kinds of objects. Typically, one patent that is about a disc drive will describe how the read/write head carriage is moved around, while another disc drive patent will concentrate on how the disc is inserted into the drive. The problem that this poses is severe in the context of MERGE, because MERGE is designed to categorize many F-trees about similar objects. Since RESEARCHER is not bound to read only about disc drives, the solution seems to be to broaden its knowledge base so that it can deal with a wide variety of physical objects that might occur in conjunction with disc drives. This would take a substantial effort, but would result in a more robust program.

6.5 What's missing in CORPORATE-RESEARCHER and RESEARCHER

Having presented the CORPORATE-RESEARCHER and RESEARCHER programs, we next compare their implementations of MERGE against the ideal scheme. In doing so, a better analysis of these programs' performance can be made. In addition, the more crucial aspects of MERGE that bear on these programs will be determined . It is unlikely that a researcher would need to implement all the features of the ideal MERGE scheme described in Section 6.2 for a particular application. Reviewing CORPORATE-RESEARCHER's and RESEARCHER's use of MERGE can serve as a case study on what parts of MERGE are needed for a given domain.

6.5.1 A review of the implementations of MERGE

Much of the program code having to do with MERGE is shared by both CORPORATE-RESEARCHER and RESEARCHER. However, they each benefit from different aspects of it to greater or lesser extents.

CORPORATE-RESEARCHER's domain is well organized and the data provided to it is very complete, so it makes limited use of MERGE's ability to fill in missing information. However, level-hopping is quite important because corporations tend to have different chain-of-command lengths in their hierarchies. Thus, in order to

recognize common sub-hierarchies in various corporations, level-hopping is essential. An F-tree in CORPORATE-RESEARCHER is itself equivalent to a chart; in most cases, few relations are needed to augment the F-tree because corporations don't usually include them on their charts. Although relations exist within corporations, they are either too complex or too subtle to chart, so CORPORATE-RESEARCHER doesn't really need MERGE's relation processing components to a great extent.

RESEARCHER, on the other hand, uses many relations in representing complex physical objects. It therefore needs, and has, a sophisticated relation representation scheme. Generalizations of relations are important and should be extended down to the primitive level of relation characteristics, as was described in Section 5.3. Level-hopping and the assumption of missing information are also important in RESEARCHER, because the textual data supplied to the system is incomplete and inconsistent. In general, RESEARCHER needs a lot more out of MERGE than does CORPORATE-RESEARCHER due to its natural language input and more complex domain of understanding.

6.5.2 Comparison to ideal MERGE

Here, we compare our two implementations of MERGE-based systems to the ideal scheme presented earlier in this chapter. The numbers used correspond to those used in Section 6.2.

Features of CORPORATE-RESEARCHER and RESEARCHER:

1. Generalization-based memory - Obviously, RESEARCHER, CORPORATE-RESEARCHER, and any other system using MERGE necessarily have this feature.

2. Dynamic memory - RESEARCHER and CORPORATE-RESEARCHER do this incremental reorganization very well. Most other MERGE-based systems would also need this feature. The only exception would be in a system that could carefully order its input so that reorganization is not needed (i.e., the subtraction and substitution operations are not needed, only the addition operation).

3. Framed-based representations - The memettes used in RESEARCHER and CORPORATE-RESEARCHER are somewhat different. The domain in which MERGE is used determines the required frame slots.

4. Parallel generalizations - This is done very effectively in both programs. Any implementation of MERGE would want to do this if learning about objects on multiple levels in the F-tree is important.

5. Inheritance - This is essential to any implementation of MERGE. It is

one of the primary ingredients in unifying representation and generalization.

6. Automatic classification - In both programs, only the top level memettes in the F-trees are kept around when not needed so that they are categorized by the system. It is possible to keep any number of memettes at any level around, or none at all. The requirements of the application should determine this.

7. Incremental learning - An automatic benefit of using MERGE.

8. Large domains - RESEARCHER is a particularly good example of this. There are over 4 million patents, hundreds on disc drives alone. Unfortunately, they don't all describe complete disc drives.

9. Massive reorganization/error correcting - Neither program does this, although it is possible given MERGE's formalism. The problem is in determining when a major reorganization is needed. It is important in domains with a large number of instances. The ability to correct errors via reorganization could be useful in both programs, particularly in RESEARCHER as its input tends to produce erroneous F-tree representations and, therefore, G-trees.

10. Process varied data - Relations are generalized in both programs. We are currently extending RESEARCHER's relation generalization capabilities to break down relation characteristics into primitives. We are also adding function generalization to CORPORATE-RESEARCHER, and property generalization to RESEARCHER. The need for such generalizations are largely dependent on the application domain.

11. Multiple inheritance - Neither RESEARCHER nor CORPORATE-RESEARCHER allow this at present (with the possible exception of ALTERNATE-VARIANT-OF slot usage). If it is known that inherited data will not conflict, then this could be added to a system by simply allowing a list of memettes to fill the VARIANT-OF slot, and changing the inheritance functions.

12. Level-hopping - The level-hopping mechanism in both programs is good. It is only needed in domains that describe hierarchies without using a standardized means for representing levels.

13. Accessible knowledge structure - RESEARCHER makes some use of the unified memory structure in disambiguating further input. CORPORATE-RESEARCH has no need to do this, since its input is unambiguously hand coded. RESEARCHER also uses its knowledge base in a question answering sub-system.

6.6 Summary

The main principle of MERGE is that the representation of hierarchies influences the generalizations that are made about them, and that these generalizations, in turn, influence future representations. This is made possible by arranging memory in terms of a hierarchy of generalizations and dynamically reorganizing it when needed.

Basic MERGE functions on a represent - generalize - represent feedback cycle, potentially increasing its knowledge base on each cycle. Using this method, generalizations are created that both categorize instance F-trees and allow missing or incomplete information to be assumed. An ideal MERGE-based system would have many more features. These include: level-hopping, the ability to correct erroneous generalizations and/or representations, massive memory reorganization when needed, and others.

CORPORATE-RESEARCHER uses a less than ideal, but better than basic, implementation of MERGE, in a system designed to understand corporate charts. It serves as a demonstration program that makes clear how MERGE works. RESEARCHER is a program that reads patent abstracts about complex physical objects using a similar version of MERGE to organize its memory. It is a natural language processing system that processes large amounts of real-world information, and thus demonstrates how various aspects of MERGE can be useful in understanding incomplete and ambiguous hierarchies.

This chapter concludes the thesis by citing directions for future research and summarizing the main points brought out in the previous chapters.

7. Conclusion

7.1 Future research

The basic principles of the MERGE scheme have been developed, at least to the point of being able to use them in functioning programs. However, there are several avenues along which this work can be extended.

The most obvious first step would be to apply MERGE-based understanding to other hierarchical domains. We have found some pluses and minuses that CORPORATE-RESEARCHER and RESEARCHER have in common. It would be useful to know how universal these findings are. In addition, some of the difficulties that RESEARCHER has in forming representations may suggest solutions when compared with similar natural language processing applications of MERGE.

We have suggested a few possible areas where a MERGE-based understanding system might be useful. Zoological taxonomies based on animal body part hierarchies seem to be an interesting and instructive domain. Input could come from textual descriptions, manual encodings, visual data from anatomical drawings, or a combination of these. The resulting knowledge structures could be compared against existing taxonomies, providing a metric to analyze MERGE's performance by

Several issues in generalization deserve further consideration. The idea of focusing generalizations around a particular object level in the instance hierarchies has been mentioned. Applying the concept of "basic objects" to MERGE's representation/generalization interaction would make it more cognitively accurate.

Another aspect of human cognition, the "aha" response, corresponds to an insightful large scale reorganization of memory in MERGE. On occasion, the G-trees that are created in a MERGE-based system can become inefficient in the sense that the branching factor is too large or too small. They may also not be the best classification of the instance objects, in that they do not closely match people's conceptions of the same objects. In addition, there may be problems that arise due to the input order dependence that MERGE-based systems have. In any of these cases, a massive reorganization of memory (as opposed to the small scale, incremental reorganization that MERGE usually performs) would solve these problems. The difficult part of doing this reorganization is recognizing when it is needed. Further research could shed some light on how to give MERGE the "aha" response.

The MERGE scheme goes a long way toward automating the intelligent understanding of hierarchies. However, some intelligence must still reside with the human system builder. In particular, a person must determine what the F-rel of a hierarchy is and develop a relation representation scheme for each new domain. Can this process be automated as well? One possible approach to solving this problem might be to build a meta-level MERGE-based system that would understand how hierarchical systems, in general, are put together.

Directions for future work also lead to basic object representation questions. We have assumed that only systems of objects structured as strict trees will be processed by a MERGE-based system. But what if this restriction is relaxed, and arbitrary networks permitted? Can MERGE be suitably modified to process such systems? If it can be, then the domains opened to application of our scheme would be increased. We believe that it can be, and that much of what has been presented in unifying representation and generalization has a more universal application.

7.2 Thesis summary

We have shown that when representations of hierarchies are stored in a GBM, the result is an enhancement of both the individual representations and the generalizations built upon them. The particular scheme used to attain this feedback, MERGE, is designed to incrementally learn by continually restructuring its knowledge base. The unification of representation and generalization in a dynamic hierarchy understanding system is the main contribution of this research.

The ubiquity of hierarchies in the real world makes this research particularly useful. MERGE provides a way to automatically classify hierarchically structured objects according to their internal organization as opposed to some artificial measure. Multiple classification hierarchies are constructed simultaneously, one for each unique sub-hierarchy in the instance objects. Thus, learning is carried out on several different objects that are part of the top-level one. Because a MERGE-based system has the capacity to capture detail to an arbitrary depth, it can potentially perform better than humans at classifying complex objects. Furthermore, since MERGE-based systems are designed to process large numbers of instances, they are suitable for use as intelligent databases.

We have used a single F-tree decomposition of a hierarchy in this thesis. Decomposing a system using one F-rel has been done mostly for pragmatic reasons, but it has not severely limited the domains that MERGE can understand. Information that is not captured by F-rel links is superimposed on the hierarchy's representation using relations. This allows partial, alternate decompositions of a hierarchical system to be captured, if necessary, along with any other data dependent on the structure of the system. Information that is independent of a

hierarchy's structure, such as properties and features, can also be included in its representation and subsequent generalizations. If necessary, two or more independent MERGE-based systems can be used for understanding a particular domain that has several equally important decompositions of its hierarchies.

Two implementations of the MERGE scheme have been developed and demonstrated. CORPORATE-RESEARCHER is a program that understands upper-level corporate organizational structures by analyzing corporate charts. RESEARCHER is a larger, natural language processing system that reads and understands patent abstracts about complex physical objects (disc drives). Taken together, they exemplify the range of hierarchical object understanding that MERGE is capable of.

The performance of these two programs is as expected. That is, CORPORATE-RESEARCHER creates generalized concepts of corporations that are similar to what people might create when shown the same charts that it processes. RESEARCHER has some difficulty producing interesting generalizations because the patent abstracts it reads vary widely in what they describe. The solution seems to lie in broadening RESEARCHER's domain of understanding so that it will find similarities among different objects related to disc drives. However, the program does a fairly good job of representing patent abstracts as augmented F-trees.

Real world implementations bring out some of the details necessary to build hierarchy understanding systems. The ability to level-hop turns out to be crucial in both systems we have built, and in any other implementation that must process incomplete or non-standardized instance objects representations. Creating a canonical scheme for representing non-fundamental relations is another important aspect of implementing a hierarchy understanding system in domains that consist of complex hierarchies. RESEARCHER's scheme for representing physical relations among parts of complex objects is an example of this. It provides a paradigm for constructing canonical relation representation schemes.

Aside from the inherent feedback between representation and generalization, a particular MERGE-based system can make use of its knowledge base for other processing. In RESEARCHER, the parser uses information in the generalization hierarchies to help in disambiguating text. The knowledge structures are also used as the basis of an intelligent information system in conjunction with an integrated question answering module.

We believe that it is necessary to unify representation and generalization in order to create intelligent information systems. This is particularly true when the goal is to understand complex phenomena. The MERGE scheme does this for hierarchically structured objects. Since complexity often takes the form of a hierarchy, our scheme should have wide applications in intelligent information systems geared toward understanding complex objects.

References

[**Abelson 73**] Abelson, R.P. The structure of belief systems. In R.C. Schank and K. Colby, Ed., *Computer Models of Thought and Language*, W. H. Freeman Co., San Francisco, 1973.

[**Angluin and Smith 82**] Angluin, D. and Smith, C.H. A Survey of Inductive Inference: Theory and Methods. Tech. Rept. 250, Yale University Department of Computer Science, 1982.

[**Barr and Feigenbaum 81**] Barr, A. and Feigenbaum, E.A.. *The Handbook of Artificial Intelligence Vol. 1.* William Kaufman, Inc., Los Altos, Calif., 1981.

[**Ben-Bassat and Zaidenberg 84**] Ben-Bassat, M. and Zaidenberg, L. "Contextual Template Matching: A distance measure for patterns with hierarchically dependent features." *IEEE Transactions on Pattern Analysis and Machine Intelligence 6*, 2 (1984), 201 - 211.

[**Birnbaum and Selfridge 81**] Birnbaum, L. and Selfridge, M. Conceptual analysis of natural language. In R. C. Schank and C. K. Riesbeck, Ed., *Inside Computer Understanding*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1981, pp. 318 - 353.

[**Bobrow and Winograd 77a**] Bobrow, D.G. and Winograd, T. "An Overview of KRL, a Knowledge Representation Language." *Cognitive Science 1* (1977), 3 - 46.

[**Bobrow and Winograd 77b**] Bobrow, D.G. and Winograd, T. Experience with KRL-0 one cycle of a knowledge representation language. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, International Joint Conference on Artificial Intelligence, 1977, pp. 213 - 222.

[**Bobrow et al. 77**] Bobrow D.G. et al. "GUS, A Frame-Driven Dialog System." *Artificial Intelligence 8* (1977), 155 - 173.

[**Brachman 79a**] Brachman, R.J. Taxonomy, descriptions and individuals in natural language processing. Proceedings of the 17th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, La Jolla, California, 1979, pp. 33 - 38.

[**Brachman 79b**] Brachman, R.J. On the epistemological status of semantic networks. In N.V. Findler, Ed., *Associative Networks*, Academic Press, New York, N.Y., 1979.

[**Buchanan and Mitchell 78**] Buchanan, B.G. and Mitchell, T.M. Model-directed learning of production rules. In D.A. Waterman and F. Hayes-Roth, Ed., *Pattern-Directed Inference*, Academic Press, New York, 1978, pp. 297 - 312.

[**Carbonell 70**] Carbonell, J.R. "AI in CAI: An artificial intelligence approach to computer-aided instruction." *IEEE Transactions on Man-Machine Systems 11*, 4 (1970), 190 - 202.

[**Carbonell 81**] Carbonell, J.G.. *Subjective Understanding: Computer Models of Belief Systems.* UMI Research Press, Ann Arbor, Michigan, 1981.

[**Chomsky 65**] Chomsky, N.. *Aspects of the Theory of Syntax.* MIT Press, Cambridge, Mass., 1965.

[**Churchman 64**] Churchman, C.W. An approach to general systems theory. In Mesarovic, M.D., Ed., *Views on General Systems Theory*, John Wiley & Sons, New York, 1964, pp. 173 - 175.

[**Cullingford 78**] Cullingford, R. Script application: Computer understanding of newspaper stories. Tech. Rept. 116, Yale University Department of Computer Science, 1978.

[**Dietterich and Michalski 81**] Dietterich, T.G. and Michalski, R.S. "Inductive Learning of Structural Descriptions: Evaluation criteria and comparative review of selected methods." *Artificial Intelligence 16* (1981), 257 - 294.

[**Fahlman 79**] Fahlman, S.E.. *NETL, a System for Representing and Using Real-world Knowledge.* MIT Press, Cambridge, Mass., 1979.

[**Fillmore 68**] Fillmore, C. The case for case. In E. Bach and R. Harms, Ed., *Universals in Linguistic Theory*, Holt, Rinehart and Winston, New York, New York, 1968, pp. 1 - 88.

[**Hayes 77**] Hayes, P.J. On semantic nets, frames and associations. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, International Joint Conference on Artificial Intelligence, 1977, pp. 99 - 107.

[**Hemenway and Tversky 84**] Hemenway, K. and Tversky, B. "Objects, Parts, and Categories." *Journal of Experimental Psychology: General 113* (1984), 169 - 1193.

[**Hendrix 79**] Hendrix, G.G. Encoding knowledge in partitioned networks. In N.V. Findler, Ed., *Associative Networks*, Academic Press, New York, N.Y., 1979.

[**Kintsch 77**] Kintsch, W.. *Memory and Cognition.* John Wiley and Sons, New York, 1977.

[**Kolodner 80**] Kolodner, J.L. Retrieval and organizational strategies in conceptual memory: A computer model. Tech. Rept. 187, Yale University Department of Computer Science, 1980.

[**Kruskal and Sankoff 83**] Kruskal, J.B. and Sankoff, D. An anthology of algorithms and concepts for sequence comparison. In Sankoff, D. and Kruskal, J.B., Ed., *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley Publishing Company, London, 1983.

[**Kuipers 75**] Kuipers, B.J. A frame for frames: Representing knowledge for recognition. In D. Bobrow and A. Collins, Ed., *Representation and Understanding: Studies in Cognitive Science*, Academic Press, New York, 1975.

[**Kuipers 77**] Kuipers, B. Modeling spatial knowledge. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, International Joint Conference on Artificial Intelligence, 1977, pp. 292 - 298.

[**Lebowitz 80**] Lebowitz, M. Generalization and memory in an integrated understanding system. Tech. Rept. 186, Yale University Department of Computer Science, 1980. PhD Thesis

[**Lebowitz 82**] Lebowitz, M. "Correcting Erroneous Generalizations." *Cognition and Brain Theory 5*, 4 (1982), 367 - 381.

[**Lebowitz 83a**] Lebowitz, M. Intelligent information systems. Proceedings of the Sixth International ACM SIGIR Conference, ACM SIGIR, Washington, DC, 1983.

[**Lebowitz 83b**] Lebowitz, M. RESEARCHER: An overview. Proceedings of the Third National Conference on Artificial Intelligence, American Association for Artificial Intelligence, Washington, DC, 1983, pp. 232 - 235.

[**Lebowitz 83c**] Lebowitz, M. Concept learning in a rich input domain. Proceedings of the International Machine Learning Workshop, Champaign-Urbana, Illinois, 1983, pp. 177 - 182.

[**Lebowitz 83d**] Lebowitz, M. "Generalization from Natural Language Text." *Cognitive Science 7*, 1 (1983), 1 - 40.

[**Lebowitz 84**] Lebowitz, M. Using memory in text understanding. Proceedings of ECAI-84, Pisa, Italy, 1984.

[**Lehnert 77**] Lehnert, W. G.. *The Process of Question Answering.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1977.

[**Lehnert 78**] Lehnert, W.G. Representing physical objects in memory. Tech. Rept. 131, Yale University, 1978.

[**Lehnert and Burstein 79**] Lehnert, W.G. and Burstein, M.H. The role of object primitives in natural language processing. Tech. Rept. 162, Yale University Computer Science Department, 1979.

[**Lindsay et al. 80**] Lindsay, R., Buchanan, B.G., Feigenbaum, E.A., and Lederberg, J.. *DENDRAL.* McGraw-Hill, New York, 1980.

[**Manheim 66**] Manheim, M.L.. *Hierarchical Structure: A Model of Design and Planning Processes.* MIT Press, Cambridge, Mass., 1966.

[**McCoy 82**] McCoy, K.F. Augmenting a database knowledge representation for natural language generation. Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, Toronto, Ontario, Canada, 1982, pp. 121 - 127.

[McKeown 82] McKeown, K. R. Generating natural language text in response to questions about database structure. University of Pennsylvania, 1982.

[Mesarovic 64] Mesarovic, M.D. Foundations for a general systems theory. In Mesarovic, M.D., Ed., *Views on General Systems Theory*, John Wiley & Sons, New York, 1964, pp. 1 - 24.

[Michalski 83] Michalski, R.S. "A Theory and Methodology of Inductive Learning." *Artificial Intelligence 20* (1983), 111 - 161.

[Michalski and Stepp 83a] Michalski, R.S. and Stepp, R.E. "Automated Construction of Classifications: Conceptual clustering versus numerical taxonomy." *IEEE Transactions on Pattern Analysis and Machine Intelligence 5*, 4 (1983), 396 - 409.

[Michalski and Stepp 83b] Michalski, R.S. and Stepp, R.E. Learning from observation: Conceptual clustering. In R.S. Michalski, J.G. Carbonell and T.M. Mitchell, Ed., *Machine Learning*, Tioga Publishing, Palo Alto, Calif., 1983, pp. 331 - 363.

[Miller 56] Miller, G.A. "The Magic Number Seven Plus or Minus Two: Some limits on our capacity for processing information." *Psychological Review 63* (1956), 81 - 97.

[Minsky 75] Minsky, M. A framework for representing knowledge. In P.H. Winston, Ed., *The Psychology of Computer Vision*, McGraw-Hill, New York, 1975.

[Mitchell 77] Mitchell, T.M. Version spaces: A candidate elimination approach to rule learning. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, International Joint Conference on Artificial Intelligence, 1977, pp. 305 - 310.

[Mitchell 82] Mitchell, T.M. "Generalization as Search." *Artificial Intelligence 18* (1982), 203 - 226.

[Noetzel and Selkow 83] Noetzel, A.S. and Selkow, S.M. An analysis of the general tree-editing problem. In Sankoff, D. and Kruskal, J.B., Ed., *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley Publishing Company, London, 1983.

[Paris 84] Paris, C. Determining the level of expertise of a user of a question answering system. Tech. Rept. 84-3-paris-1, Tech. Rept., Columbia University Department of Computer Science, 1984.

[Pattee 73] Pattee, H.H. Physical basis and origin of hierarchical control. In H.H. Pattee, Ed., *Hierarchy Theory: The Challenge of Complex Systems*, MIT Press, Cambridge, Mass., 1973, pp. 71 - 108.

[Quillian 68] Quillian, M.R. Semantic memory. In M. Minsky, Ed., *Semantic Information Processing*, MIT Press, Cambridge, Mass., 1968.

[Roberts and Goldstein 77] Roberts, R.B. and Goldstein, I.P. The FRL manual. Tech. Rept. 409, MIT AI Laboratory, 1977.

[Rosch et al. 76] Rosch, E. et al. "Basic Objects in Natural Categories." *Cognitive Psychology 8*, 3 (1976), 382 - 437.

[Rosenbloom and Newell 83] Rosenbloom, P.S. and Newell, A. The chunking of goal hierarchies: A generalized model of practice. Proceedings of the International Machine Learning Workshop, Champaign-Urbana, Illinois, 1983, pp. 183 - 197

[Sacerdoti 75] Sacerdoti, E.D. A structure for plans and behavior. Tech. Rept. 109, SRI International, 1975.

[Schank 72] Schank, R.C. "Conceptual Dependency: A theory of natural language understanding." *Cognitive Psychology 3*, 4 (1972), 532 - 631.

[Schank 75] Schank, R.C. *Conceptual Information Processing.* North Holland, Amsterdam, 1975.

[Schank 80] Schank, R.C. "Language and Memory." *Cognitive Science 4*, 3 (1980), 243 - 284.

[Schank 82] Schank, R.C. *Dynamic Memory: A Theory of Reminding and Learning in Computers and People.* Cambridge University Press, New York, 1982.

[Shlichta 69] Shlichta, P.J. Overlap in hierarchical structures. In Whyte, L.L., Wilson, A.G. and Wilson, D., Ed., *Hierarchical Structures*, American Elsevier, New York, 1969, pp. 138 - 143.

[Simon 73] Simon, H.A. The organization of complex systems. In H.H. Pattee, Ed., *Hierarchy Theory: The Challenge of Complex Systems*, MIT Press, Cambridge, Mass., 1973, pp. 1 - 27.

[Simon 81] Simon, H.A. The architecture of complexity. In H.A. Simon, Ed., *The Sciences of the Artificial*, MIT Press, Cambridge, Mass., 1981.

[Sussman and Steele 80] Sussman, G.J. and Steele, G.L. Jr. "CONSTRAINTS - A Language for Expressing Almost-Hierarchical Descriptions." *Artificial Intelligence 14* (1980), 1 - 39.

[Tversky 77] Tversky, A. "Features of similarity." *Psychological Review 84*, 4 (1977), 327 - 352.

[Wasserman 84] Wasserman, K. Understanding Hierarchically Structured Objects. Tech. Rept., Columbia University Department of Computer Science, 1984.

[Wasserman 85] Wasserman K. "Physical object representation and generalization: A survey of semantic-based natural language processing programs." *AI Magazine 5*, 4 (1985), 28 - 42.

[Wasserman and Lebowitz 83] Wasserman, K. and Lebowitz, M. "Representing Complex Physical Objects." *Cognition and Brain Theory 6*, 3 (1983), 333 - 352.

[Webber 75] Webber, R.A. Static principles of organizational design. In R.A. Webber, Ed., *Management*, Richard D. Irwin, Inc., Homewood, Illinois, 1975.

[White 63] White, K.K. *Understanding the Company Organization Chart.* American Management Association, New York, 1963.

[Wilensky 78] Wilensky, R. Understanding goal-based stories. Tech Rept. 140, Yale University Department of Computer Science, 1978.

[Wilks 74] Wilks, Y. Natural language understanding systems within the AI paradigm. A survey and some comparisons. Tech. Rept. 237, AI Laboratory Stanford University, 1974.

[Winograd 72] Winograd, T. *Understanding Natural Language.* Academic Press, New York, 1972.

[Winograd 75] Winograd, T. Frame representation and the declarative/procedural controversy. In D. Bobrow and A. Collins, Ed., *Representation and Understanding: Studies in Cognitive Science*, Academic Press, New York, 1975.

[Winston 72] Winston, P.H. Learning structural descriptions from examples. In P.H. Winston, Ed., *The Psychology of Computer Vision*, McGraw-Hill, New York, 1972.

[Winston 80] Winston, P.H. "Learning and Reasoning by Analogy." *Communications of the Association for Computer Machinery 23*, 12 (1980), 689 - 703.

[Woods 75] Woods, W.A. What's in a link: Foundations for semantic networks. In D. Bobrow and A. Collins, Ed., *Representation and Understanding: Studies in Cognitive Science*, Academic Press, New York, 1975.

Appendix A. F-tree Matching

In this appendix we present the algorithm used to match F-trees in CORPORATE-RESEARCHER and RESEARCHER. This algorithm serves as the metric used to decide how similar two F-trees are in order to determine where in memory a new instance F-tree should be located. In addition, this algorithm is used to determine the structure of the generalized F-tree that is formed from comparing a new instance F-tree to an existing F-tree. (Although, this feature will not be discussed here.)

We have found this to be an effective algorithm for use in MERGE. However, it must be emphasized that this is only one of many possible algorithms that a MERGE-based system could use. The reader may find work done by other researchers helpful in developing similar F-tree matching algorithms. Of particular interest is work in: [Noetzel and Selkow 83] that presents a procedure for determining the minimum number of operations needed to permute one tree (F-tree) into another, [Tversky 77] that describes numerical similarity measures, and [Kruskal and Sankoff 83] which presents a survey of work dealing with various methods for comparing sequences of data

A.1 Overview

Generalization in RESEARCHER and CORPORATE-RESEARCHER is done by making binary comparisons of F-trees. The program code for matching F-trees has two inputs: the old F-tree and the new F-tree. The old F-tree is one that already exists as part of memory. It may either be a generalized F-tree (one that was created by the program) or it may be an instance F-tree (one that was input to the program). The new F-tree is not yet in memory. The only connection that the new F-tree has with memory is that its nodes are variants of the same concepts that nodes in other F-trees are. That is, all F-trees are defined in terms of a common set of initial concepts. These concepts are the root nodes of G-trees (see examples in Chapter 6).

The basic matching procedure works by recursive descent through the input F-trees. At each level, a doubly nested loop is run through trying to match all children nodes in one F-tree against all children nodes in the other F-tree. When a leaf node is reached in either the new F-tree or the old F-tree it is compared against its corresponding node. If they are both variants of the same initial concepts then some preset number of points is given to this correspondence. (Other possibilities are discussed below.) This score is passed back up one level and used to decide which are the best child-child matches to choose. Once this "best" matching of children is picked, the scores for each child-child match are summed and the matching of the parent nodes is summed into this.

The score returned from MATCH for the entire F-tree is the result of summing the scores for each matched pair of nodes. However the scoring process is somewhat more complex in that each level in the F-trees is given a different weight. The higher level nodes are more important than are the lower level nodes and thus the grade given to a match of them is higher. In addition, relations that a particular memette is involved in also contribute to the total score. Finally, when a node has components that do not exist in the other (new or old) F-tree these count negatively toward the match score. The effect of doing this is to not match similar memettes if the children in one of them would have to be deleted via the subtraction operation unless outweighed by a different matching.

Because F-trees in a given domain are usually not standardized some way of finding correspondences between nodes that are not on the same level in both the new F-tree and the old F-tree is needed. Level-hopping is used to find these correspondences. NULL# memettes are inserted in either or both the old F-tree and the new F-tree to achieve level-hops. The insertion of a NULL# memette counts negatively toward the match score. This effectively restricts the use of NULL# memettes, so that too many levels are not inserted.

A.2 The basic algorithm

Level-hopping is accomplished using the function HOP-MATCH (see Figure A-1). HOP-MATCH calls MATCH using variations on the input F-trees. For example, HOP-MATCH calls MATCH with no inserted NULL# memettes, it then calls MATCH with a NULL# memette inserted in front of the root node of the old F-tree, it then tries the same but using the new F-tree. It also tries other combinations by inserting NULL# memettes immediately after the root node, etc. After trying all these combinations it chooses the resultant configuration with the highest matched score.

```
PROCEDURE HOP-MATCH(memette-a,memette-b)
 BEGIN
    IF [(memette-a is unitary) AND (memette-b is unitary)]
       OR [(memette-a is NULL#) AND (memette-b is NULL#)]
       THEN RETURN MATCH(memette-a,memette-b);
    ELSE
       RETURN
        MAX(MATCH(memette-a,memette-b),
            MATCH(memette-a,(insert a NULL# before memette-b))
            MATCH(memette-a,(insert a NULL# after memette-b))
            MATCH((insert a NULL# after memette-a),memette-b)
            MATCH((insert a NULL# before memette-a),memette-b));
 END
```

This is the top-level function in the F-tree matching process. It calls MATCH to do all the work. NULL# memettes are inserted before and after each node in both F-trees. The result of the best configuration is picked (by the MAX function). This is how level-hopping is achieved.

Figure A-1: HOP-MATCH - the level-hopping procedure.

MATCH (see Figure A-2) is the basic matching code which works as described above. However, instead of calling itself recursively it calls HOP-MATCH on the subtrees of the F-tree it is currently analyzing. In this way, level-hopping occurs at all levels in the F-trees.

Several factors determine what will be the "best" match of nodes in the new F-tree with nodes in the old F-tree. The scoring system captures all of these factors as a single number. As was mentioned above, points are added or subtracted from this score on three occasions: 1 - when two leaf nodes are compared, 2 - when two parent nodes are compared, 3 - when a parent node has unmatched children.

```
PROCEDURE MATCH(memette-a,memette-b)
 BEGIN
    !level!:=!level!-1;
    score:=COMPARE(memette-a,memette-b);
    IF memette-a is UNITARY
       THEN
        BEGIN
           !level!:=!level!+1;
           RETURN !level! * score;
        END;
    ELSE
       BEGIN
          DO FOR all x:=(F-children of memette-a)
           BEGIN
              best-child-match:=0;
              DO FOR all y:=(F-children of memette-b)
                 best-child-match:=
                    MAX(best-child-match,HOP-MATCH(x,y));
              score:=score+best-child-match;
              REMOVE the best-child matched from
                     memette-b's F-children;
           END;
          !level!:=!level!+1;
          RETURN !level!*[score +
                          !extra-parts-penalty! *
                          (the number of unmatched F-children)];
       END;
 END
```

MATCH compares the new F-tree rooted at memette-a against the new F-tree rooted at memette-b. The score is computed by adding the value returned by COMPAREing the root nodes to the sum of the best matches of the trees' F-children. The variable !level! is used to weight the value returned at each level (higher levels are worth more). !Level! must be initialized in the calling procedure to be the greater of the depth of the new F-tree or the old F-tree.

Figure A-2: MATCH - the basic algorithm.

The function COMPARE (see Figure A-3) returns the score computed by comparing the two memettes that it is passed as arguments. The score returned depends on what the memettes are ultimately variants of, whether one or both of them are NULL# memettes, and any relations the memettes might be involved in. Variables are set in the calling procedure that are used for each of these cases (they are indicated by !xxxxxx!). Another variable, !at-level! is incremented and decremented in MATCH to keep track of how deep in the recursion the program is, and is used as a multiplier factor for the !xxxxxx! variables. COMPARE-RELATION is called by COMPARE to compute the value added for relation matches.

The score for any subtree is computed by summing the scores for all the matched children of the subtree together with the score for matching the root of the subtree. Any unmatched children are subtracted from the score according to the value that !extra-parts-penalty! has.

```
PROCEDURE COMPARE(memette-a,memette-b)
 BEGIN
     CASE OF
        VARIANT-OF(memette-a)=VARIANT-OF(memette-b):
                sum:=!same-variant-of-match!;
        IS-NULL#(memette-a) OR IS-NULL#(memette-b):
                sum:=!null-real-match!;
        OTHERWISE:
                sum:=!unknown-part-match!;
     ENDCASE;

     DO FOR all x:=(relations that involve memette-a)
        BEGIN
           best-relation-match:=0;
           DO FOR all y:=(relations that involve memette-b)
              best-relation-match:=
                 MAX(best-relation-match,
                     COMPARE-RELATION(x,y));
           sum:=sum+best-relation-match;
           REMOVE the best-relation matched from
                  the relations that memette-b is in;
        END;
     RETURN sum;
 END
```

The COMPARE function evaluates how two individual memettes compare. If they are ultimately variants of the same initial concept then they are considered to match better than if they are not. This is true only if the value of !same-variant-of-match! is greater then !unknown-part-match! This function returns a score that is dependent on both what the memettes are variants of and how they are used in relations. COMPARE-RELATION is a simple function that looks inside relation frames and returns a non-zero value if it finds some commonalities between x and y.

Figure A-3: COMPARE - the lowest-level evaluation function.

A.3 Scoring variables

In this section we discuss each of the variables of significance to the F-tree matching algorithm. The values that have been used to generate the examples in this thesis are shown in parenthesis after the name of each variable. We have found the program to be fairly insensitive to the values of these variables. However, it is sensitive to the relative ordering of their values. For example, !same-variant-of-match! > !unknown-part-match! > !null-real-match! produces different results than !unknown-part-match! > !same-variant-of-match! > !null-real-match!.

- !same-variant-of-match! (+7) - This is the value assigned to a memette matching where both memettes (old and new) are found to ultimately be variants of the same initial concept (by following the VARIANT-OF links all the way up the G-tree). In the program used for the examples in this thesis, this value is assigned regardless of how many G-tree

VARIANT-OF links need to be followed. However we have experimented with using a percentage of this value that depends on how many links must be followed.

- !null-real-match! (-2) - When a real memette (i.e., non-NULL# memette) is matched against a NULL# memette this is the value assigned to their match. Using a negative value for !null-real-match! has the effect of penalizing the use of level-hopping. This technique is a heuristic that limits the number of levels that will be inserted to give a good match.

- !unknown-part-match! (+1) - When two memettes are matched and they are not ultimately variants of the same initial concept, then an UNKNOWN# memette is created in their generalized F-tree. The value of this variable is assigned to a match of memettes in these cases. Its value should be less than the value of !same-variant-of-match!, but still positive compared to the value of !null-real-match!.

- !extra-parts-penalty! (-3) - This is used when there are unmatched F-children at any level in the F-tree. The number of such children multiplies this variable, and this product is added into the score returned from MATCH. It should have a value that is negative relative to both !unknown-part-match! and !same-variant-of-match!. Its purpose is to push the algorithm into matching as many memettes as possible.

- !level! - This variable is initialed to be the depth of the deepest F-tree (new or old). It is then decremented at each call to MATCH, and incremented upon return. It is used as a multiplying factor for all of the above variables. Its purpose is to allow memette matches higher in the F-tree to count more strongly than memette matches lower in the F-trees