Learning Plan Schemas from Cases

A Dissertation Presented
by
Robert S. Williams

COINS Technical Report 90-94

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

LEARNING PLAN SCHEMAS FROM CASES

A Dissertation Presented

By

ROBERT S. WILLIAMS

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1990

Computer and Information Science

©Copyright by Robert Stuart Williams 1990
All Rights Reserved

This research supported by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract no. N00014-87-K-0238, the Office of Naval Research, under a University Research Initiative Grant, Contract # N00014-86-K-0764 and NSF Presidential Young Investigators Award NSFIST-8351863, and by the Advanced Research Projects Agency of the Department of Defense, monitored by the Air Force Office of Scientific Research under Contract No. F49620-88-C-0058.

DEDICATION

This thesis is dedicated to my wife, Lynn Pocock-Williams.

ACKNOWLEDGMENTS

Wendy Lehnert has been to me an exemplar of the way AI research should be done. While encouraging me to think for myself, she has taught me much about developing models of intelligence through writing programs. She has given my work focus and coherence.

My other committee members, Edwina Rissland, Paul Utgoff, and Clive Dym, have all left their marks on this thesis. Edwina has brought with her a deep understanding of case-based reasoning, allowing her to evaluate my contribution to the field. Paul has encouraged me to write with precision and to substantiate my claims. Clive has played the role of AI outsider, challenging me to make this work more comprehensible to a general audience.

Thanks to Claire Cardie, Tony Reish, Ellen Riloff, David Skalak, Kishore Swaminathan, and Stephan Wermter for participating in the program writing experiment. Thanks also to Steve Bradtke, Tim Howells, and Dave Haines for their graphics routines, and to Dave Lewis and Bruce Leban for useful comments.

Finally, I would like to thank my parents, Lee and Jerry, for instilling in me the desire to learn. And most importantly, I thank my wife Lynn — without her, there would be no thesis.

ABSTRACT

This thesis addresses the problem of learning in the context of case-based reasoning (CBR). More specifically, we examine the limitations of and possible uses for generalization in the process of solving problems using cases.

In some sense, every CBR system that acquires new cases dynamically can be said to learn, since the new cases enable the system to improve its performance over time. We usually, however, think of learning as involving some sort of change in representation. Most CBR work that deals with constructing new representations is in the area of indexing, i.e., learning descriptions for cases that enable them to be retrieved in appropriate situations.

Our interest in learning includes learning generalized case descriptions, but there are certain techniques for such generalization that we feel are inappropriate. Specifically, there is a body of work that advocates using dynamically generated discrimination networks to index cases. We claim that, if one assumes a parallel retrieval mechanism, it is unnecessary to use explicit generalizations of this sort to index cases.

In addition to learning generalized case descriptions, we are interested in generalizations of the cases themselves. We demonstrate that such generalizations can be of help both in the process of completing past cases to fit current situations and in the analysis of failed solution attempts. We present a technique for creating such generalizations that is well suited to case-based problem solving. The technique, called analytic concept creation, can be used in generalizing both cases and case descriptions.

In order to carry out our explorations, we have developed a model for case-based problem solving, called RECODER. RECODER is a model that attempts to integrate the various phases of case-based problem solving, including retrieval of relevant cases, completion of past cases to fit current situations, debugging when errors occur, and reorganization (adding new cases and generalizing). The model has been implemented in a system called TA, which writes and debugs small LISP programs.

Table of Contents

Acknowledgments iv

Abstract v

List of Tables x

List of Figures xi

Chapter

1. Introduction 1

1.1 Overview 1
1.2 Work in Case-Based Reasoning 3
1.3 A Critique, a Proposal, and a Technique 4

1.3.1 Generalizations and Indexing: A Critique 4
1.3.2 Generalizations in Problem Solving: A Proposal 4
1.3.3 A Technique for Generalizing Cases 5

2. Selected Relevant Literature: A Review 7

2.1 Introduction 7
2.2 MBRTalk 9
2.3 CYRUS 11
2.4 The PERSUADER 12
2.5 CHEF 15

2.5.1 Emphasis 16
2.5.2 Representation 17
2.5.3 Process vs. Vocabulary 18

2.6 SOAR 19

3. THE RECODER MODEL 21

3.1 Introduction 21
3.2 Problem Solving as Planning 21
3.3 Types of Knowledge Used by RECODER 22

3.3.1 Cases 22
3.3.2 Prior Knowledge 22
3.3.3 Advice 23

3.4 RECODER's Components 24

3.4.1 Retrieval 25
3.4.2 Completion 27
3.4.3 Debugging 29
3.4.4 Reorganization 31

4. THE TA SYSTEM 34

4.1 Introduction 34
4.2 Knowledge Sources in TA 36

4.2.1 Hierarchical Relations 36
4.2.2 Axiomatic Plans 41
4.2.3 An Example of Plan Application 48

4.3 TA in Action 51

4.3.1 Retrieval 51
4.3.2 Completion 53
4.3.3 Debugging 55
4.3.4 Reorganization: Creating Difference Maps 56
4.3.5 Reorganization: Creating Schemas and Chunks 57
4.3.6 A Final Example 60

5. MEMORY ORGANIZATION 62

5.1 Introduction 62
5.2 The Indexing Problem 62

5.2.1 Work in Matching and Assimilation of Indices 63

5.3 Matching and Assimilation of Indices in RECODER 66

5.3.1 Retrieval 66

5.3.1.1 Feature Selection 67
5.3.1.2 Index Matching 69

5.3.2 Reorganization 71

5.4 Integrating Case Memory and Background Knowledge 72

6. GENERALIZATIONS IN PROBLEM SOLVING 75

6.1 Introduction 75
6.2 Generalizations in Plan Modification 75

6.2.1 Work in Analogical Problem Solving 76
6.2.2 Solving Problems in RECODER 76
6.2.3 Using Generalizations to Focus Problem Solving 82
6.2.4 Conclusions 87

6.3 Generalizations in Debugging 87

6.3.1 Work in Diagnosing Plan Failures 88
6.3.2 Debugging in RECODER 90
6.3.3 Using RECODER to Analyze Buggy Plans 93

6.3.3.1 The Experiment 93
6.3.3.2 The Programs 94
6.3.3.3 TA's Analysis 97

6.3.4 Conclusions 99

7. ANALYTIC CONCEPT CREATION 101

7.1 Introduction 101
7.2 The Learning Problem 101
7.3 Related Learning Work 102
7.4 Generalization in RECODER 103

7.4.1 Types of Generalization in RECODER 103
7.4.2 Creating Schemas and Schema Descriptions in TA 104
7.4.3 Schema Creation and Explanation-Based Learning 107

7.5 Comparison of Clustering Techniques 109

7.5.1 TA's Generalizations 109
7.5.2 Generalizations from UNIMEM and COBWEB 109

7.6 Conclusions 115

8. COMPLETING THE MODEL 116

8.1 Introduction 116
8.2 State of the Art 117

8.2.1 Retrieval 117
8.2.2 Completion 117
8.2.3 Debugging 118
8.2.4 Reorganization 118

8.3 A Brief History of TA 118

8.3.1 TA.1 119
8.3.2 Retrieval 120
8.3.3 Debugging 121
8.3.4 Reorganization 122
8.3.5 Comparing TA.1 and TA.2 123

8.4 Further Work 126

APPENDICES 129

A. COMPLETION TIME COMPARISON SPECIFICATIONS 130
B. RETRIEVAL IN RECODER AND UNIMEM 132
B.1 Introduction 132
B.2 Phase One 133
B.3 Phase Two 144

BIBLIOGRAPHY 150

LIST OF TABLES

Table — Page

2.1 Representations in CHEF and RECODER. 18

6.1 Total number of plan steps and average time spent in completion. . . 83
6.2 Time spent computing inconsistencies. 83
6.3 Times spent in plan modification. 85
6.4 Comparing false hits in TA with and without schemas. 87

7.1 Concepts found via clustering algorithms: structured features bias . . 110
7.2 Concepts found via clustering algorithms: flat features bias 111
7.3 Specifications and corresponding program names. 113
7.4 Differences in categorization between COBWEB and UNIMEM. . . . 114

8.1 Associations between LISP code and naive model. 124

B.1 Comparing UNIMEM's best and worst performance to RECODER. . 136
B.2 Comparing retrieval with original bias, at a precision of 0.35. 146
B.3 Comparing retrieval with positional bias, at a precision of 0.5. 148
B.4 Comparing retrieval with "letters-after" bias, at a precision of 0.35. . 149

LIST OF FIGURES

Figure		Page
3.1	A planning situation represented as a network of concepts.	24
3.2	The RECODER architecture.	25
3.3	RECODER's retrieval component.	26
3.4	A second planning situation.	26
3.5	RECODER's completion component.	27
3.6	RECODER's debugging component.	30
3.7	RECODER's reorganization component.	32
3.8	A generalized planning situation.	32
4.1	High-level concepts about programs in TA.	37
4.2	The `if` specification concept.	37
4.3	Specification concepts dealing with collections.	38
4.4	More specification concepts.	38
4.5	A specification represented as a network of concepts in TA.	39
4.6	The **not** program concept.	39
4.7	Single input program concepts.	40
4.8	Two-input program concepts.	40
4.9	Three-input program concepts.	40
4.10	A network representation for `LAT?`.	52
4.11	A network representation for `ALL-EMPTY?`.	52
5.1	A network representation for `LAT?`.	68
5.2	Case memory in TA	70
5.3	Cases linked to hierarchical relations	73
5.4	The effects of learning on prior knowledge	74
6.1	A network representation for a specification.	78
6.2	A second network representation for a specification.	80
6.3	Speedup gained from completing schemas as opposed to plans.	84

6.4 Speedup gained in examining schemas as opposed to plans. 85

7.1 A case description. 106
7.2 A second case description. 107
7.3 A generalized description. 108
7.4 Concept descriptions generated by TA. 110
7.5 UNIMEM tree for program specifications. 112
7.6 COBWEB tree for program specifications. 112
7.7 Further subdivisions of the COBWEB tree. 113

8.1 The architecture for TA.1. 119
8.2 The architecture for TA.2 (i.e., RECODER). 120
8.3 TA.1's retrieval component. 121
8.4 TA.1's debugging component. 122
8.5 TA.1's reorganization component. 123

B.1 History for growth of UNIMEM tree {8, 1}, trained on set 2. 134
B.2 History for growth of UNIMEM tree {2, 1}, trained on set 2. 135
B.3 Comparing RECODER with UNIMEM best and worst. 136
B.4 "Goodness" history for a generalization tree. 137
B.5 Similarity history for "tail." 138
B.6 Similarity history for "vault." 138

CHAPTER 1

INTRODUCTION

1.1 Overview

We are concerned with providing a model of problem solving in a domain initially unfamiliar to the problem solver. We begin our investigation into such a model with the following observations:

- People seem to use past experience when solving problems, when such experience is available.
- The more familiar one becomes with a given problem domain, the better one becomes at solving problems in that domain.
- Even experts make mistakes.

What will a model of problem solving that is consistent with the above observations look like? Such a model must address the following questions:

- How are we reminded of past experiences?
- How do we relate past experiences to current problems?
- How do we deal with mistakes we might make in problem solving?
- How do we improve at solving problems?

We address these questions from within the paradigm of *case-based reasoning* (CBR) [Rissland and Ashley, 1987, Hammond, 1988, Kolodner *et al.*, 1985] and the closely related *memory-based reasoning* paradigm [Stanfill and Waltz, 1988]. Past problem-solving episodes are stored as *cases*, and are retrieved based on similarities to the current problem situation. A case is *completed* by using prior knowledge to yield something more applicable to the current situation. If a mistake is made, an attempt is made to remember cases of similar failures and how those failures were corrected. Learning takes place by remembering and generalizing cases.

The main focus of this thesis is the role that generalization plays in the process of solving problems using cases. We are interested in finding both limitations of and possible uses for generalization. The thesis can be seen to consist of three components: a *critique*, a *proposal*, and a *technique*.

1. *critique*: The primary use of generalization in most CBR work to date has been to organize cases for ease of retrieval. We claim that the type of generalization most often associated with case retrieval is not significantly better than organizational schemes that do not use explicit generalization.

2. *proposal*: While generalizations do not necessarily aid in the retrieval of cases, they can be helpful in the problem-solving process itself. We identify two areas of that process where such utility can be demonstrated.

3. *technique*: We introduce a technique for generalizing cases that is uniquely suited to case-based problem solving.

At the heart of the thesis is a model for case-based problem solving that enables generalizations to be made. We call our model RECODER, since it contains:

- A component to REtrieve past cases from memory;
- A component to COmplete past cases, yielding something more applicable to the current situation;
- A component for DEbugging if mistakes have been made;
- A component to Reorganize memory by adding and generalizing cases.

The name also reflects the case-based philosophy underlying the model: rather than generating solutions to problems from first principles, old solutions are recoded to fit new problems.

In order to explore various ramifications of the RECODER model, we have implemented parts of that model in computer programs. The most significant implementation is a program called TA, which writes and debugs small LISP programs in the style of [Friedman, 1974]. TA is not a serious threat to other automatic programming systems, at least in its present form, since so far it has been used to generate only very small programs. We have selected the domain rather for the insights it allows us into case-based problem solving. In particular, programming has two benefits:

1. It is a *problem-solving* domain. Problems in the programming domain can often be cleanly represented as specifications, and the ultimate solutions can be cleanly represented as programs. Further, it is easy to tell when a solution is viable: a solution (i.e., a program) is viable if it works according to the specification.

2. It is a *case-based* domain. New programs are often written using existing programs as models. This method is stressed in [Friedman, 1974], where the reader is often encouraged to use older programs as guides in writing new ones.

Thus, writing programs is a real-world example of case-based problem solving, and further, one whose results can be independently verified.

1.2 Work in Case-Based Reasoning

There are many areas of active research in CBR. Much of this work addresses one or more of the following three broad problems:

- *The indexing problem.* How is case memory arranged so that good cases can be found efficiently?
- *The analogy problem.* How are past cases to be used in solving the problem at hand? Work addressing this problem includes arguing from precedents [Ashley and Rissland, 1986, Ashley, 1988], adapting past cases to fit current problems [Alterman, 1988], and dealing with failures resulting from incorrect adaptations [Hammond, 1987].
- *The learning problem.* How are new cases assimilated?

We are particularly interested in the learning problem in the context of CBR. In some sense, every CBR system that acquires new cases dynamically can be said to learn, since the new cases enable the system to improve its performance over time [Plan, 1989]. This is learning in a very restricted sense, though. We usually think of learning as involving some sort of change in representation, that is, as a *transformation* of information provided by an environment into some new form [Michalski, 1986]. Most CBR work that deals with constructing new representations is in the area of indexing, and falls into two broad categories. One group of work, exemplified by [Kolodner, 1984, Kolodner *et al.*, 1985], emphasizes building discrimination networks [Feigenbaum, 1963] based on similar features of cases; these networks are traversed in order to retrieve potentially relevant cases. A second body of work is exemplified by [Barletta and Mark, 1988]; in this work, indices for cases are created using techniques derived from explanation-based learning [Mitchell *et al.*, 1986, DeJong and Mooney, 1986].

Our interest in learning includes learning generalized case descriptions, but we are also interested in generalizations of the cases themselves. There are three main issues that we examine in the body of this thesis, each loosely related to one of the problems stated above:

1. *Techniques for indexing cases.* As mentioned above, there is a body of work that advocates using dynamically generated discrimination networks to index cases. We claim that, if one assumes a parallel retrieval mechanism, it is unnecessary to use explicit generalizations of this sort to index cases. This is the *critique* component of the thesis.

2. ***Using generalizations in solving problems.*** We look at two aspects of the analogy problem, using past cases and dealing with failures, and describe ways in which generalizations of past cases can be of help. This is our ***proposal*** for the use of generalizations.

3. ***Learning new cases and case descriptions.*** There are two aspects to case-based problem solving: it is an approach to problem-solving, and it makes use of cases. We present a generalization technique that takes advantage of both aspects. This approach allows us to generalize both cases and case descriptions. This is the ***technique*** component of the thesis.

1.3 A Critique, a Proposal, and a Technique

1.3.1 Generalizations and Indexing: A Critique

As mentioned earlier, the predominant use of generalization in CBR systems has been to index cases. Much of this work has centered around building discrimination networks of case descriptions to organize case indices. In this thesis, we argue that such an approach is not necessary for efficient case retrieval. We contend that a flat case organization, coupled with a parallel partial matching algorithm, will retrieve cases at least as relevant as those retrieved using a discrimination network organization. In addition, the parallel algorithm will be time efficient, and the flat memory organization will be space efficient.

To support this claim, chapter 5 describes a memory organization designed to have the functionality given by discrimination networks, but using a flat memory and a parallel retrieval algorithm. Appendix B offers some experimental support of the merits of this organization by comparing it with UNIMEM [Lebowitz, 1986], a dynamic discrimination network approach to indexing cases. The experiments in appendix B indicate that for the simple purpose of retrieving relevant cases from memory, RECODER performs at least as well as UNIMEM.

1.3.2 Generalizations in Problem Solving: A Proposal

There may still be situations in which it is desirable to generalize cases and case descriptions. We contend that generalized cases can aid in solving problems and in diagnosing buggy plans. In both situations, we demonstrate our claims using plan schemas.

As formulated in RECODER, a necessary part of completing plans is finding inconsistencies between the current state and the state associated with a retrieved plan. Any steps relating to these inconsistencies must be rewritten. Thus, when completing plans, it is necessary to find inconsistencies and to check each step in a retrieved plan for those inconsistencies. These actions become unnecessary when

completing plan schemas, since all inconsistencies have been abstracted out. It is only necessary to find steps that have been generalized, and make these steps concrete. In this and other ways, the planning process becomes more efficient when using generalizations.

To illustrate this, we shall describe experiments evaluating the utility of generalizations in RECODER. The experiments compare a version of RECODER that creates case generalizations with one that does not. We show that completing case generalizations is more time efficient than completing actual cases, due to the overhead of computing inconsistencies. We also show that using generalizations can actually save space, since not all new cases need to be stored in their entirety.

Plan schemas can also be used in diagnosing failed plans, as a way of putting those plans in context. In effect, this turns diagnosis into a classification task. A failed plan is classified as a perturbation of an existing plan schema.

To illustrate the use of plan schemas in failure diagnosis, we shall describe an experiment that uses schemas created by RECODER to analyze programming errors taken from human subjects. In essence, the schemas serve as a way to give context to the errors.

1.3.3 A Technique for Generalizing Cases

Case-based problem solving is just that: an approach to solving problems that makes use of cases (past solutions to similar problems). In considering how to generalize cases, then, case-based problem solving gives us two natural constraints:

1. As a case-based task, we are naturally constrained to generalize on the basis of similarities between cases.

2. As a problem solving task, we are naturally constrained to generalize on the basis of solving problems successfully.

Using these constraints as a guideline, we arrive at a technique for generalizing cases called *analytic concept creation.* This technique is related to concept clustering, in that it creates generalized descriptions of cases by generating clusters based on similarity of features; it is an analytic technique in that it uses solutions to problems (created using domain knowledge) to control the generalization process.

This technique is used for creating generalized cases (schemas), as described in the section 1.3.2. But in order to find the appropriate schema for a particular problem, schemas must be indexed by generalized case descriptions, and so the technique is used for this also.

To show the benefits of using analytic concept creation to created generalized descriptions, we shall describe an experiment in which we compare a set of descriptions created by RECODER with concept hierarchies created by two non-analytic clustering techniques, UNIMEM [Lebowitz, 1986] and COBWEB [Fisher, 1987]. We

will show that the set of descriptions generated by RECODER is better suited to indexing schemas than either of the non-analytic techniques.

CHAPTER 2

SELECTED RELEVANT LITERATURE: A REVIEW

2.1 Introduction

The aim of this chapter is to position the RECODER model with respect to related work in case-based reasoning and learning.

In case-based reasoning (CBR), current situations are reasoned about by appeal to similar past situations. Three general areas of CBR can be distinguished:

- *Argument Formation* (e.g., [Ashley, 1988]): Given a fact situation and a position, substantiate the position by appealing to similar situations.
- *Planning* (e.g., [Hammond, 1988]): Given a statement of a problem, find a plan that solves a similar problem and modify it to fit the current problem.
- *Weak-Methods Search* (e.g., [Korf, 1985]): Given a state in a state space, find and apply a sequence of operators that when applied to a sufficiently similar state, results in a state closer to a goal.

Argument formation is distinguished by its use of multiple cases for a single situation. Both planning and weak-methods search use a single case for a given situation, though complex situations may be decomposed into collections of simpler situations. Argument formation and planning both assume the existence of background knowledge to understand cases in the context of the current situation. Weak-methods search is distinguished from planning by virtue of its paucity of background knowledge. In planning, this background knowledge is used to modify old plans to fit current situations; since no such knowledge exists in weak-methods search, it is assumed here that operators (or operator sequences) will be used without modification.

RECODER is a model of case-based planning. Case-based planning differs from traditional approaches to planning (e.g., [Fikes and Nilsson, 1971]) in that it creates new plans by modification of old, similar plans rather than from scratch. Accordingly, all case-based planners must address how to *retrieve* similar plans, and how to *modify* those plans to fit current situations. RECODER additionally addresses how to *learn* from past problem-solving experiences.

According to [Carbonell, 1989], current research in machine learning can be divided into four areas:

- *Inductive learning*: Given a sequence of instances, build a concept description which characterizes or discriminates between those instances.
- *Analytic learning*: Given a performance task, a small set of exemplars (often a single one) and some background knowledge, build "operational" concepts which improve task performance.
- *Genetic learning*: Learn to classify instances of a concept using techniques "inspired by a direct analogy to mutations in biological reproduction ... and Darwinian natural selection."
- *Connectionist learning*: Given training sets of representative instances of a class, learn to recognize these and other instances of the class by readjusting weights in a fixed-topology network.

Both inductive and analytic learning can be incorporated into models of case-based planning. Insofar as new plans are derived via analytic means, analytic learning is exhibited. Insofar as several concrete are used to generate abstract plans and planning situations, inductive learning is exhibited. The RECODER model incorporates both of these learning paradigms.

RECODER is made up of four components: Retrieval, Completion, Debugging, and Reorganization. The retrieval component is responsible for selecting plans for situations that are similar to the current situation. This is done by comparing the current situation to previous situations, and selecting the most similar situation. Each situation *indexes* a plan; a plan is selected by selecting the situation that indexes it (we refer to situations that index plans as plan *descriptions*). The completion and debugging components together comprise RECODER's modification capabilities.[1] The reorganization component is responsible for learning from planning experience. Learning is accomplished by assimilating new plans (derived via analytic means) and by (inductive) generalization. Generalization is performed both on plans themselves, and on the situations in which plans apply (i.e., on plan descriptions).

The remainder of this chapter will illustrate the various facets of RECODER by comparing it with five related systems. RECODER's model of case retrieval will be compared with retrieval in MBRTalk [Stanfill and Waltz, 1986] and CYRUS

[1]There is not universal agreement on the terms used to describe the process of modifying plans. Most researchers would agree that what they do is an example of *modification*, but they may have other names for specific techniques, e.g., *adaptation* [Alterman, 1988], *repair* [Hammond, 1988], *analogical transformation* [Carbonell, 1983], etc. RECODER employs two modification techniques: *completion*, in which background knowledge is used to transform old plans to fit new situations, and *debugging*, in which plans found to be in error are corrected.

[Kolodner, 1984]. Case modification in RECODER will be compared to modification in the PERSUADER [Sycara, 1987] and CHEF [Hammond, 1988]. RECODER's learning capabilities will be compared with chunking in SOAR [Laird *et al.*, 1986, Laird *et al.*, 1987].

2.2 MBRTalk

MBRTalk is an exemplar of the *memory based-reasoning* paradigm [Stanfill and Waltz, 1986, Stanfill and Waltz, 1988]. Memory-based reasoning (MBR) is an attempt to describe all reasoning in terms of memory retrieval. MBRTalk is an implementation of MBR in the domain of word pronunciation.

Memory in MBRTalk consists of a large number of *records*, which correspond to cases in CBR. Each record contains a number of *predictors* (case descriptions) and *goals* (plans). In the simplest version of MBRTalk, a record contains seven predictors, each referring to a letter in a word, and one goal, a pronunciation. A word containing n letters would have n records in memory: one for each letter in the word. The record for the ith letter in a word would contain as predictors that letter, the three letters preceding it, and the three letters following it. The goal for the letter would be the phoneme representing the pronunciation of the letter in the word.

The task of pronouncing a word is a three-step process. In the first step, the word is decomposed into a sequence of records, one for each letter in the word. In the second step, memory is searched to retrieve the closest match for each of these records. The final step composes the goals of each of the retrieved records into a sequence of phonemes.

It should be clear from the above description that MBRTalk is in fact an example of weak-methods search (as defined above). Modification of results is not possible in the system; the individual results (i.e., the phonemes) are treated as atomic entities. The performance of the system is entirely dependent on the ability of memory search to retrieve the best match to a given record. This ability stems from two sources: a *parallel retrieval algorithm*, ensuring efficient retrieval, and a *similarity metric*, allowing for correct results.

In MBR, it is assumed that memory search can be performed in parallel. Retrieval proceeds by broadcasting to each case a record, having each case compute its similarity to the record in parallel, and selecting the best matching cases.

The crucial step in the MBR process is computing the similarity to a given case. Stanfill and Waltz describe several possible metrics for computing this. The first, which they call an *overlap metric*, is simply counting the number of matching predictors in the case and the current record.

The overlap metric has several problems when used in the context of MBR,

the most significant of which is that all predictors in a record are treated equally. This problem is addressed by modifying the overlap metric so that predictors are weighted. The weight for a given predictor is determined by taking the square root of the sum of the squares of the frequencies of the possible goal values for that predictor. The resulting metric is called the *weighted feature metric.*

The weighted feature metric is still too strict for Stanfill and Waltz, in that "it is based on the precise equality of the values of the predictor fields" [Stanfill and Waltz, 1986]. To solve this problem, the weighted difference metric is modified to allow for similarity of predictor values. Similarity between two features is computed by calculating the frequencies of the goal values for each of the features, subtracting one from the other, squaring the results, and summing over all goal values. This new metric is called the *value difference metric.*

It should be apparent that the value difference metric is significantly more complex than the original overlap metric. This complexity is needed for MBR, since the lack of explicit background knowledge forces the similarity metric to play the role of background knowledge. The metric as described above (and in [Stanfill and Waltz, 1988]) is demonstrated to work reasonably well in the word pronunciation domain, but there is no evidence that it will be equally applicable in other domains. In fact, Stanfill and Waltz motivate their modifications to the original metric by appealing to problems that arise in word pronunciation; different domains may yield different problems that call for different modifications.

RECODER is similar to MBR in its assumption that case retrieval is best thought of as a parallel process, with each case simultaneously comparing itself to the current situation. In fact, RECODER's retrieval algorithm follows the same general outline as MBRTalk, in that a current situation is broadcast to all cases, each case computes its similarity to the situation, and the most similar case is selected as the best match.

RECODER differs from MBR in its emphasis on retrieval. For MBR, retrieval comprises the bulk of the reasoning process; there is very little work done to a case once it has been retrieved. For RECODER, retrieval is only the first step of the reasoning process; the significant work is in the modification of old cases and the assimilation of new ones. For this reason, RECODER does not need an elaborate similarity metric; in fact, the metric it uses is essentially Stanfill and Waltz's overlap metric. The reason that RECODER can make effective use of this metric will MBR cannot is intimately related to the primary difference between case-based planning and weak methods search. Weak methods search assumes that the case base will contain an exact match to the current situation, and that the purpose of the similarity metric is to recover that match. Case-based planning assumes that the case base does *not* contain an exact match to the current situation, and uses the similarity metric to find the most appropriate candidate for subsequent modification.

2.3 CYRUS

CYRUS [Kolodner, 1984] is an early implementation of Schank's theory of Dynamic Memory [Schank, 1982]. The task of CYRUS to answer questions based on events in the public life of then secretary of state Cyrus Vance.[2]

The core of CYRUS is a memory of these events, organized by structures called *E-MOPs*. An E-MOP consists of two parts: the *norms* of the events that it organizes, and the *differences* between those events. The norms of an E-MOP are those features that are shared by all events organized by the E-MOP; the differences, those features that differ. Differences act as *indices* to more specific E-MOPs and to specific events, by including pointers to those E-MOPs and events. Questions are given as sets of features. These features are matched with the most general E-MOP, and subsequently with objects pointed to by those differences of the E-MOP that match features of the question. This matching process is continued until no further matches are possible, at which time the appropriate E-MOPs or events are returned. The hierarchy of E-MOPs is generated automatically as new events are encountered, by following the question-answering process outlined above with the features of a new event replacing the "question." At the points at which "answers" are found, new E-MOPs are created based on similarities between the new event and the events in memory.

CYRUS's hierarchy of E-MOPs, along with the associated retrieval and generalization process, could be viewed as retrieval and learning components of a hypothetical case-based planner; in fact, later case-based problem solving systems [Kolodner *et al.*, 1985] take this approach. But CYRUS itself lacks one important ingredient of a case-based planner: there are no plans. In effect each event in CYRUS's memory corresponds to what we have been calling a case *description*. An event is little more than a collection of features, any one of which may be used as either a norm or a difference in any given E-MOP. Once an event has been retrieved (as the answer to a question), it is merely presented to the asker. There is no additional problem-solving task for which a retrieved event can be used.

Even as a model of retrieval, CYRUS leaves something to be desired. There is no guarantee that the features of a question will correspond in a natural way to features as they appear in the E-MOP hierarchy. This leads to the necessity in CYRUS for *elaboration*, Kolodner's term for providing additional features to be used in search. A large part of Kolodner's description of CYRUS in [Kolodner, 1984] is given over to discussion of the various kinds of heuristics used for elaboration. One example of such a heuristic is: "to infer the country an event might have taken place in, use the participants' country of residence, country they habitually travel to, or their nationality" ([Kolodner, 1984], p. 61).

An additional problem with a dynamically generated hierarchical memory is

[2] A second version of CYRUS answered questions about Vance's successor, Edmund Muskie.

that there is no general way to control the rate at which the memory grows. That this is indeed a problem in CYRUS can be inferred from the following comment, in regard to a retrieval component derived from CYRUS: "Such a memory may, however, explode in size as increasing numbers of generalizations are stored in it" ([Sycara, 1987], p. 90).

RECODER's retrieval component offers an alternative to hierarchies of events for organizing cases. In RECODER, events (cases) are not indexed in hierarchies of generalizations. Instead, search is accomplished by broadcasting a "question" directly to all events in parallel; each event determines (in parallel) its degree of match to the "question," and the event with the maximum degree of match is retrieved as the best match. This mechanism does away with both of the problems mentioned above. Since there is no hierarchy of generalizations separating questions from events, there is no danger of features deep in the hierarchy being hidden from corresponding features in a question. And since explicit generalizations are not constructed, there is no danger of the size of memory exploding.

2.4 The PERSUADER

The PERSUADER [Sycara, 1987] is a problem solver in the domain of labor mediation. The PERSUADER is a hybrid system, mixing case-based problem solving techniques with analytic methods (which will not be discussed further). Sycara decomposes the case-based capabilities of the PERSUADER into five parts as follows:

1. Retrieve appropriate cases from memory.
2. Select the most appropriate case(s) from those retrieved.
3. Construct a "ballpark" solution.
4. Evaluate the "ballpark" solution for applicability to the current case.
5. Modify the "ballpark" solution appropriately.

Cases are stored via a memory organization derived from CYRUS [Kolodner, 1984], so step 1 is much like retrieval from CYRUS. Step 2 applies some domain-specific heuristics to select cases similar with respect to features deemed to be "important." The features used in these first two steps are derived from a *dispute* structure, a partial description of a case that lists such things as the disputants involved (generally a company and a labor union), the geographic location of the company and the union, and the industry that the company is involved in.

Once a case has been selected, it is modified via steps 3, 4, and 5. The content of a case is a *contract* between a company and a union. This contract represents a resolution of a past *impasse* between the company and the union. A contract is transformed into a "ballpark" solution according to a set of heuristics relating features of disputes to features of impasses. "Ballpark" solutions are evaluated by *critics*, which take into account features of the current situation such as the financial situation of the company, the structure of the bargaining unit (e.g., whether most of the employees are of a certain age group), and political considerations of the company and the union. If evaluation deems it necessary, the "ballpark" solution is modified further, either by using past cases or via repair heuristics associated with each critic.

In the PERSUADER, disputes and impasses together correspond to case descriptions in RECODER: disputes correspond to general state information, and impasses correspond to goals of the planner. Contracts in the PERSUADER correspond to modifiable plans in RECODER, and heuristics for evaluating and modifying contracts correspond to RECODER's axiomatic plans.

The only mechanism for generalization in the PERSUADER is to be found in the automatic creation of hierarchies due to the CYRUS-style memory organization. There is no generalization of cases themselves (i.e., contracts), and the generalization that is done is based solely on similarity of surface features. By contrast, RECODER generalizes both cases and case descriptions, and its generalizations are based on the success or failure of the case modification process.

To get RECODER-like generalization in the PERSUADER, successful contracts would need to be generalized. These contract schemas would be indexed, not be CYRUS-style generalizations, but by descriptions that were generalized, like the contract schemas, in response to the process of generating successful contracts. Further, for the contract schemas to be useful, their generalized portions would need to be linked somehow to corresponding portions of descriptions. These links could be determined by examining the heuristics used to make particular modifications.

As an example of how RECODER-style generalization might be accomplished in RECODER, consider the following simplified example.[3] Assume that the PERSUADER knows about the following case:

[3] This example is intended only as an illustration of how the PERSUADER might make use of RECODER-style generalization. It does not, nor is it intended to, do justice to the case modification capabilities of the PERSUADER.

DESCRIPTION:
  INDUSTRY: Transportation
  COMPANY FINANCES: Good
  UNION DEMAND: Strict seniority
  UNION DEMAND: Large wage increase
CONTRACT:
  Seniority based on skill
  Medium wage increase

Now, suppose a new mediation problem is presented:

DESCRIPTION:
  INDUSTRY: Transportation
  COMPANY FINANCES: Fair
  UNION DEMAND: Strict seniority
  UNION DEMAND: Large wage increase

Let us say that the PERSUADER retrieves the stored contract, and modifies it by using the heuristic: IF company finances are fair, THEN give only a small increase. This results in the new contract:

CONTRACT:
  Seniority based on skill
  Small wage increase

The generalization of the case would then be:

DESCRIPTION:
  INDUSTRY: Transportation
  COMPANY FINANCES: *Financial rating*
  UNION DEMAND: Strict seniority
  UNION DEMAND: Large wage increase
CONTRACT:
  Seniority based on skill
  Some wage increase (amount determined based on finances)

The description and the contract have been generalized to cover the descriptions and contracts for the old and new situations. In addition, the part of the contract describing the wage increase has been annotated to correspond with the part of the description describing company finances, since the finances were used to determine the wage increase.

2.5 CHEF

CHEF [Hammond, 1988] is a case-based planner in the domain of recipe design that emphasizes avoidance of past planning failures. CHEF is presented with a set of *goals* (including type of dish and ingredients to be used) for which a new plan (i.e., recipe) is to be constructed. CHEF views planning as a six stage process:

1. *Anticipate failures.* Past planning failures are stored in a network of failure nodes. CHEF uses a marker passing algorithm on this network to determine whether any past failure is likely to occur given the current set of goals. If so, a goal to avoid the failure is created.

2. *Retrieve a plan.* Past plans are stored in a discrimination network, indexed by goals. The network is searched for a plan with as many goals as possible that match the current set of goals. The discrimination network is ordered so that more important goals appear near the root of the network and less important goals appear nearer the leaves. This ordering is given by a *goal value hierarchy.*

3. *Modify the plan.* Plans are modified to fit the current set of goals via *modification rules* and *critics.* Modification rules are stored in a table indexed by type of dish and ingredient, and specify an appropriate sequence of recipe steps. Critics are special purpose rules for dealing with specific ingredients or interacting goals. CHEF is supplied with an initial set of critics, and additional ones are learned as failures are repaired.

4. *Repair the plan.* The modified plan is executed via a program that simulates the preparation of a dish. If a failure is encountered (in the form of either the absence of desired goals or the presence of unwanted results), CHEF *explains* the failure and *repairs* it. Explanations are used to index *planning TOPs,* which characterize specific failures in more general planning terms. Each planning TOP has stored with it a *repair strategy,* to recover from the failure.

5. *Assign credit for failures.* Each planning TOP also contains the set of features which caused the associated general failure to occur. These features are used to index specific failures in the failure network.

6. *Store the plan.* The corrected plan is stored in the planning discrimination network, indexed by its goals and the failures it has corrected.

In many ways, CHEF is quite similar to RECODER. The planning process is very similar: in both systems, plans are retrieved, modified using background knowledge, and assimilated into existing case bases. There are, however, three important differences between CHEF and RECODER.

1. ***Emphasis.*** RECODER emphasizes generalization; in CHEF, the emphasis is on failure avoidance.

2. ***Representation.*** RECODER strives for a small set of general-purpose mechanisms. In CHEF, there are a large number of special-purpose mechanisms.

3. ***Process vs. vocabulary.*** Hammond believes that the vocabulary used for problem solving is independent of the process by which the problem is solved; we claim that process can influence vocabulary.

Each of these differences will be discussed below.

2.5.1 Emphasis

CHEF's primary contribution is as a model of failure avoidance in case-based reasoning; as such, much of the system is concerned with reasoning about plan failures. There are 17 planning TOPs, each describing different ways in which plans might fail. Associated with each planning TOP are strategies for recovering from failures, as well as sets of features for credit assignment. In order to retrieve planning TOPs, CHEF is provided with a set of *explanation questions*, the answers to which provide indices for retrieving planning TOPs. Specific failures are stored in a network on which marker passing is performed to help avoid past failures.

Dealing with failures is a much simpler process in RECODER. Much of the machinery used in CHEF to deal with plan failures has very little to do with case-based reasoning per se, so there is little motivation to include this machinery unless modeling plan failure and recovery is a principle concern. In RECODER, it is not. We have provided RECODER with a mechanism for dealing with failures that is case-based in nature (difference maps). We have not attempted to provide a model for failure recognition or repair in the absence of prior cases. When RECODER has no prior cases to deal with a particular plan failure, it asks for advice.

Our primary aim with RECODER is to examine the nature of generalization in case-based planning. According to [Riesbeck and Schank, 1989], a fundamental type of case is the ***ossified case***, a case that has been used in so many different circumstances that many details have been abstracted away, and what remains resembles a rule. One way to look at our work with RECODER is as an exploration of the ways in which ossified cases can emerge from the planning process:

1. ***Schemas.*** Problems that can be solved in similar ways give rise to solution ***schemas.***

2. ***Chunks.*** Special-purpose routines that may apply in several situations are remembered as ***chunks.***

3. *Difference Maps.* Errors that occurred, and ways that those errors can be fixed, are remembered as *difference maps.*

Hammond does not provide CHEF with any capabilities for plan generalization. In fact, he claims that such generalization is pointless: "To [generalize plans] would be to lose information that could be used again, without gaining anything in terms of more general applicability" ([Hammond, 1988], p. 49). We disagree with this claim on both counts. First, no information need be lost through generalization, since the abstracted information can be stored elsewhere. In RECODER, generalizations of cases are stored as *schemas*, and information abstracted from schemas is stored elsewhere as *chunks* (alternately, there is nothing to prevent one from storing both a generalization and a specific case). Second, one of the principle claims of this thesis is that something is indeed gained from creating plan schemas from cases: schemas can make planning more efficient and can add explanatory power.

2.5.2 Representation

The second difference between CHEF and RECODER has do to with representation. RECODER has essentially four types of knowledge available to it: cases, axiomatic plans, hierarchical relations, and advice. In CHEF, practically every function of the system uses its own knowledge representation.

Table 2.1 compares representations for various functions in CHEF to representations for similar functions in RECODER.[4] As an example of Hammond's multiplicity of representations, note that goal abstractions, goal features (both used in retrieving cases) and role specifiers (used in finding correspondences between old plans and new goals) require three different representations in CHEF, but can all be represented in RECODER with hierarchical relations.

As another example of CHEF's proliferation of representations, consider the column in table 2.1 titled "Salient Features." This term, due to [Kolodner, 1989], refers to features of a planning situation that should be given priority when choosing plans. Kolodner describes salient features as follows: "If memory has done a good job of recording its experiences, they can be used to tell us which features or previous events led to the choice of particular solutions or solution methods and which features were responsible for success or failure in those cases. These features are the *salient features* of previous cases..." In CHEF, features to be given priority in choosing plans are determined in advance via its goal value hierarchy. In RECODER, there is no need to create a separate representation to capture the notion of salient feature: salient features are learned through the process of generalizing plans and their descriptions. A plan schema description gives the set of features

[4]Except for failure avoidance, these are functions that both CHEF and RECODER can perform. RECODER can be extended to handle failure avoidance through the use of difference maps.

Table 2.1: Representations in CHEF and RECODER.

Feature	*CHEF*	*RECODER*
Indexing	D-Net	Parallel
Avoidance	Failure Net	Cases
Modification	Mod Table	Plans
Recovery	Strategies	Advice
"IS-A"	Abstractions	Relations
"HAS-PART"	Features	Relations
Salient Features	Value Hierarchy	Cases
Detection	Explanations	Advice
Binding	Role Specifiers	Relations
Learned Repairs	Critics	Cases
Environment	Simulator	Lisp

necessary to create a plan of a given type, while abstracting out features that are irrelevant to that type of plan.

2.5.3 Process vs. Vocabulary

CHEF's goal value hierarchy also comes into play when examining the influence of process on vocabulary. According to Hammond, process and vocabulary are distinct issues: "The theoretical thrust is not centered around [the] implementations. It is instead centered around the representations that these processes use" ([Hammond, 1988], p. 71). This distinction is made, among other reasons, to de-emphasize the choice of a discrimination network to index cases. Unfortunately, the use of the discrimination network as a mechanism for partial matching of goals forces Hammond to consider the order in which various goals should be compared in the discrimination process. This leads to a need to represent knowledge about priorities of various types of goals, realized in CHEF's goal value hierarchy.

In our view, there is a relation between process and vocabulary: as seen above, process can influence vocabulary. We see RECODER's parallel retrieval algorithm as preferable to a discrimination network approach, since it leads to fewer unnecessary assumptions. In particular, we do not need the equivalent of a goal value hierarchy to represent relations between goals.

2.6 SOAR

SOAR [Laird *et al.*, 1986, Laird *et al.*, 1987] aspires to be an "architecture for general intelligence." Tasks in SOAR are formulated as finding desired states in *problem spaces*; hence, all tasks take the form of heuristic search. According to the authors, "the adoption of a problem space as the fundamental organization for *all* goal-oriented symbolic activity (called the *Problem Space Hypothesis*) is a principle feature of SOAR" ([Laird *et al.*, 1987], p. 5).

Another distinctive feature of SOAR is the idea, carried to its extreme, of *automatic subgoaling*. According to this idea, all problem solving goals are created by the architecture in responses to *impasses*,[5] i.e., places where problem solving cannot continue.

SOAR contains two primary repositories of information: *production memory* and *working memory*. Production memory contains the sum of background knowledge that SOAR brings to a particular task; working memory contains the complete processing state for solving a particular problem at any given point. Working memory is divided into three areas: *objects*, *preferences*, and a *context stack*. The context stack is essentially a hierarchy of goals yet to be solved. Objects are pointed to by goals in the context stack; they are the objects involved in the problem being solved. Preferences are structures that determine relations between objects with respect to slots in the context stack.

Processing in SOAR consists of two stages. The first stage, called *elaboration*, Uses productions from production memory to add preferences and objects to working memory. Each production takes the form:

IF C_1 and C_2 and ... and C_m THEN add A_1, A_2, ..., A_n.

Each C_i is a *condition* that examines working memory, while each A_i is an *action* that adds objects or preferences to memory. Elaboration continues until there are no productions able to fire.

After elaboration, the second stage commences. This stage is a *decision procedure* that determines which slot in the context stack should have its context replaced, and by which object. This stage begins by gathering and interpreting available preferences. If there is a preference for a particular object at a particular slot, that slot is modified with the object. Otherwise, there is an *impasse*, and a new subgoal is added to the context stack. The elaboration/decision cycle continues until the original goal has been solved.

The principle difference between SOAR's view of problem solving and RECODER's can be found in two hypotheses of SOAR's called, respectively, the *uniform elementary representation hypothesis* and the *production system hypothesis*.

---

[5]Not to be confused with impasses in the PERSUADER.

According to the first hypothesis, there is but a single elementary representation for declarative knowledge; according to the second, production systems are the appropriate representation of long-term knowledge. Under these hypotheses, all reasoning must be viewed as the application of productions to working memory.

RECODER takes a different view of the reasoning process than does SOAR. In RECODER, there is no one elementary representation for knowledge, and thus there can be no single reasoning method. RECODER supposes at least two knowledge representations: prior cases and axiomatic plans. Reasoning is a combination of case retrieval, in which cases similar to the current situation are found, and case modification, in which axiomatic plans are applied to old cases to make them fit the current situation. In this view, the SOAR single-knowledge representation can be seen as a limiting situation: if RECODER were to retrieve and modify an empty case, it would be acting as if it were not reasoning from cases at all. But in RECODER, this limit is to be avoided. Cases represent reasoning processes that need not be repeated by the reasoner, resulting in more efficient reasoning. Moreover, the reasoning embodied in cases need not have been carried out by the case-based reasoner at all.

This last point is at the heart of the difference between SOAR and case-based planning of the sort embodied in RECODER. In SOAR, the problem solver must contain all knowledge necessary to solve from scratch all problems presented to it. In case-based planning, it is not necessary for the problem solver to contain the knowledge necessary to solve a given problem from scratch: some of this knowledge may be *implicit* in a case used as a basis for modification.

SOAR contains a learning mechanism, *chunking*, which allows it to compile chains of production applications into single productions, and later use these single productions to obtain the same results as would be obtained using the longer chains. These *chunks*, once assimilated, allow SOAR to perform *as if* the knowledge used in creating them did not explicitly exist. But even though the original knowledge is not needed when reasoning with chunks, it must yet explicitly exist: "Chunking necessarily implies that there exists some way to attain goals before the knowledge has been successfully assimilated (i.e., before it has been chunked)" ([Laird *et al.*, 1987], p. 49). With a case-based planner, goals need not be attainable in the absence of cases. The cases *implicitly* encapsulate the knowledge needed to attain their associated goals.

Chapter 3

The RECODER Model

3.1 Introduction

This chapter will describe the RECODER model of case-based problem solving, and should give the reader enough of an idea of that model to follow the main points of this thesis. Chapter 4 provides an in-depth look into TA, a computer implementation of RECODER in the domain of writing computer programs. The bulk of that chapter can be skipped on first reading, but its introduction should be read to get an idea of TA's approach to automatic programming.

RECODER is a model of case-based problem solving; that is, it is a model in which cases of past solutions are used in solving current problems. There are four parts to RECODER:

1. Retrieval: a case that solves a problem similar to the current problem is retrieved.

2. Completion: the old solution is modified to fit the current problem.

3. Debugging: if the solution fails, mistakes are found and corrected.

4. Reorganization: solutions are generalized and incorporated into existing knowledge.

The main focus of this thesis is the generalization done in the reorganization component, but the components cannot easily be separated out — to understand the way in which generalizations are made, it is necessary to understand the other components as well.

3.2 Problem Solving as Planning

In the RECODER model, problem solving is characterized as a planning problem. A problem to be solved will be presented as some state information and some

goals to be achieved; a solution will be a plan, accounting for the state information, that achieves the goals.

In this chapter, we will draw examples from a toy domain based on planning to ride the subway (modeled after [Alterman, 1988]), since it is perhaps more intuitive than the domain of programming. Assume that one wanted to ride a subway in a specific subway system (we will use the Bay Area Rapid Transit, or BART, system). The state information for this problem is the environment of the local subway station (we will refer to this as the "subway situation"). The goal is to be at some destination subway station. A plan to get from the local station to the destination station is:

1 Buy a BART ticket at the ticket machine (some form of payment is needed for this).

2 Go through the turnstile at the station, using the ticket (the turnstile returns the ticket).

3 Ride the train.

4 Exit through the turnstile (using the ticket) at the destination station.

3.3 *Types of Knowledge Used by RECODER*

There are three broad types of knowledge available to RECODER: cases, prior knowledge, and advice.

3.3.1 Cases

Cases are retrieved by the retrieval component, used by the completion and debugging components, and added and modified by the reorganization component. There are four types of cases: ***plans***, ***chunks***, ***difference maps***, and ***schemas***. Plans stored as cases are composed of the axiomatic plans to be described in the following section. Chunks and schemas arise from the completion process, to be described in section 3.4.2. Difference maps arise from debugging, described in section 3.4.3.

3.3.2 Prior Knowledge

Prior knowledge, used by the completion component, is knowledge not gained directly through experience solving problems in the domain being learned. This could be knowledge codified from much experience in another domain ("common sense" knowledge), or it could be knowledge gained through reading or being told ("domain theory"). RECODER's prior knowledge comes in two forms:

- *Hierarchical relations*: This is knowledge relating concepts to one another. There are two types of relations: *isa* (as in (*isa* BART-situation subway-situation)[1]) and *has-part* (as in (*has-part* BART-situation BART-machine)[2]
- *Axiomatic plans*: These are plans that are treated as axioms. They may have been learned from a text-book or teacher (e.g., a plan for checking the atomicity of an expression in LISP by writing `(ATOM` *expression*`)`), or they may be plans that have proved so useful in so many prior situations that they have assumed an axiomatic role (e.g., a plan for buying a ticket from a ticket booth by first presenting some form of payment to a person in the booth and then receiving a ticket).

Figure 3.1 shows a set of hierarchical relations between concepts in the domain of riding the subway. In this figure, as in the remainder of the thesis, the $\Rightarrow$ symbol denotes an "isa" relation and the $\rightarrow$ symbol denotes a "has-part" relation (the direction of the arrows is for readability, and does not affect their meaning). Thus, $x \Rightarrow y$ can be read as "x is a y" (or "x has a parent concept called y"), while $x \rightarrow y$ can be read as "x has a part (or component) called y." $x \overset{*}{\Rightarrow} y$ can be read as "either $x = y$ or there exists a z such that $x \Rightarrow z$ and $z \overset{*}{\Rightarrow} y$." $x \overset{z}{\rightarrow} y$ assumes a concept C such that $x \Rightarrow C$ and $C \rightarrow z$, and is a shorthand for $x \Rightarrow C \rightarrow z \Leftarrow y \leftarrow x$, which can be read as "$y$ plays the role of z in x." This composite relation can be referred to as the *role-play* relation. Thus, the figure says that a `BART-situation` is a kind of `subway-situation`. The figure assumes that a `subway-situation` has at least three components: a `ticket-booth`, a `turnstile`, and some form of `payment`.[3] In a `BART-situation`, the role of `ticket-booth` is played by a `BART-machine` (which is a `mechanized-booth`, a kind of `ticket-booth`), the role of `turnstile` is played by a `BART-turnstile` (which is a `farecard-turnstile`, a kind of `turnstile`), and the role of `payment` is played by `money` (assumed to be an example of `payment`).

3.3.3 Advice

Advice is used solely by the debugging component, and can be thought of as knowledge coming from a source outside of the problem solver in the course of attempting to solve a problem. There are two types of advice available to RECODER:

- Evaluation oracle: For determining if a plan worked.

---

[1] I.e., "a BART-situation is a kind of subway-situation."

[2] I.e., "a component of a BART-situation is a BART-machine." *Isa* and *has-part* relations are modeled after the kind of structured inheritance found in KODIAK [Wilensky, 1987]. They are also similar to the abstraction/packaging hierarchies in [Schank, 1982] and [Riesbeck and Martin, 1985].

[3] Thus, in each of the three *role-play* relations $x \overset{z}{\rightarrow} y$ shown in figure 3.1, `subway-situation` is the concept C for which $x \Rightarrow C$ and $C \rightarrow z$.

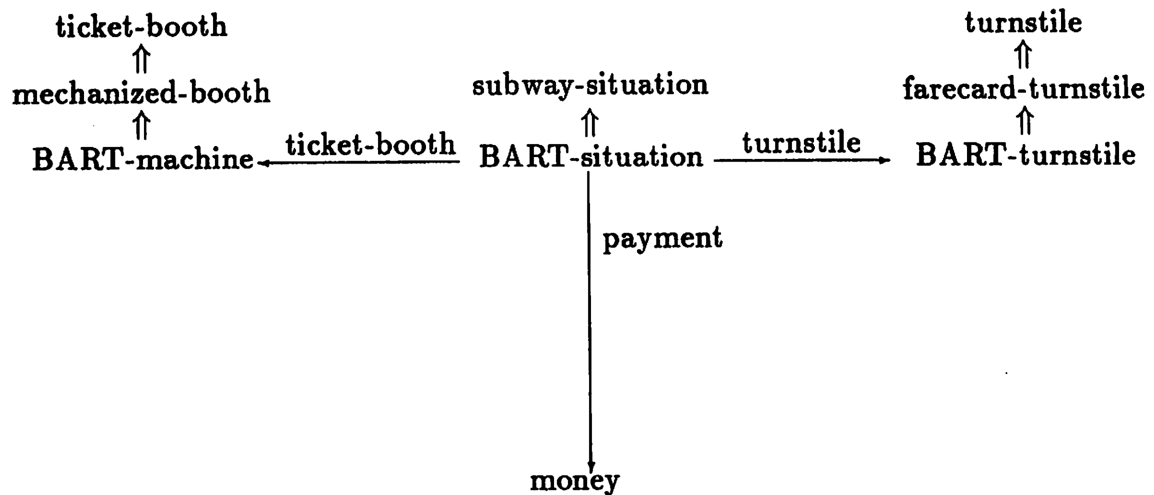

Figure 3.1: A planning situation represented as a network of concepts.

- Hints: For helping to recover from failures, e.g., by informing the problem solver which parts of a buggy program are buggy.

In an actual implementation of RECODER, it is necessary to specify the sources of advice. In both the subway domain discussed in this chapter and the program-writing domain that comprises the rest of the thesis, hints are assumed to be given by a person interacting with the system. The role of evaluation oracle for the subway domain is also played by a person interacting with the system. In the program-writing domain, the role of evaluation oracle is played by the combination of a LISP interpreter that runs generated programs and a person who supplies expected results for those runs. The program-writing system then compares the results of the LISP interpreter to the expected results.

3.4 RECODER's Components

RECODER is divided into four components: *retrieval*, *completion*, *debugging*, and *reorganization*. The knowledge available to each of the components is shown in figure 3.2. In the figure, rectangles denote the various components of RECODER, and ovals denote knowledge sources. Arrows denote the direction of the flow of knowledge: arrows from ovals to rectangles denote inputs to components, and arrows from rectangles to ovals denote outputs of the components. Outputs are not shown in figure 3.2, but will be shown in figures illustrating individual components.

The following sections will describe each of the components of RECODER in turn, illustrating the components with examples from the subway domain.

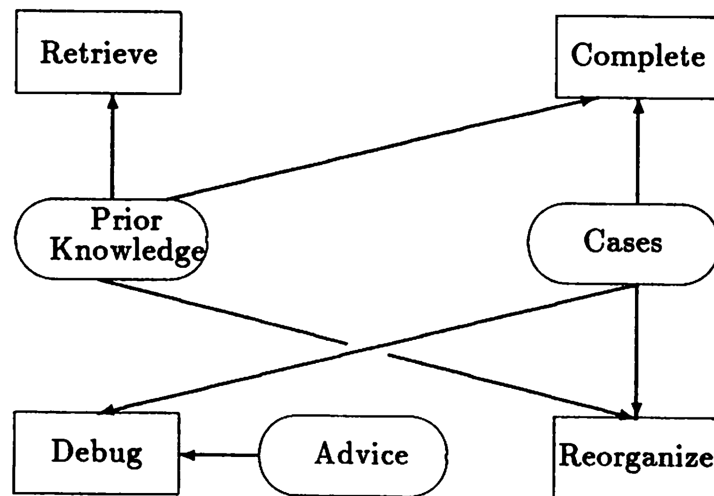

Figure 3.2: The RECODER architecture.

3.4.1 Retrieval

The retrieval component finds cases similar to the problem at hand. Retrieval uses hierarchical relations to determine which indices to use for search. Figure 3.3 illustrates the knowledge used in retrieval.

For example, consider our subway domain, and the plan shown earlier to ride a BART subway:

1 Buy a BART ticket at the ticket machine (some form of payment is needed for this).

2 Go through the turnstile at the station, using the ticket (the turnstile returns the ticket).

3 Ride the train.

4 Exit through the turnstile (using the ticket) at the destination station.

This plan is a plan for the situation shown in figure 3.1, and would thus be indexed by it. Suppose that we now wish to determine a plan for riding a New York subway; i.e., we are faced with the situation shown in figure 3.4. The retrieval component would then try to match the new situation with a situation already in memory. In this case, the BART situation (figure 3.1) would match, since both situations are kinds of `subway-situation`, and the components of both situations likewise share common ancestors.[4] Since the BART situation indexes the BART plan, that plan would be used as a first approximation for the New York situation.

[4]More detail about matching in the retrieval component is given in the next chapter.

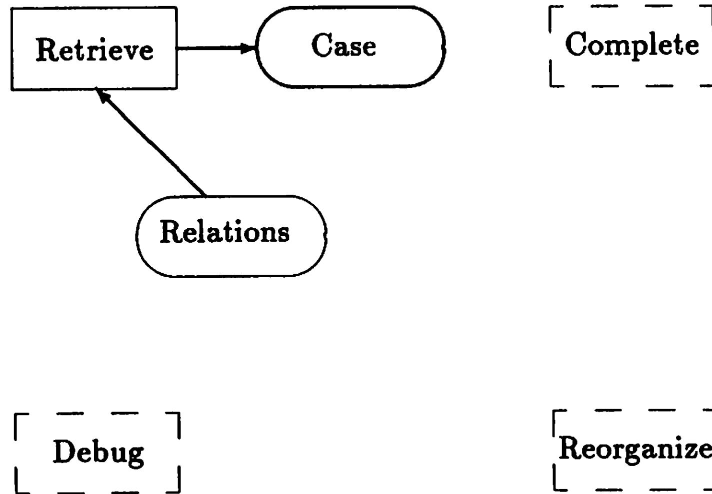

Figure 3.3: RECODER's retrieval component.

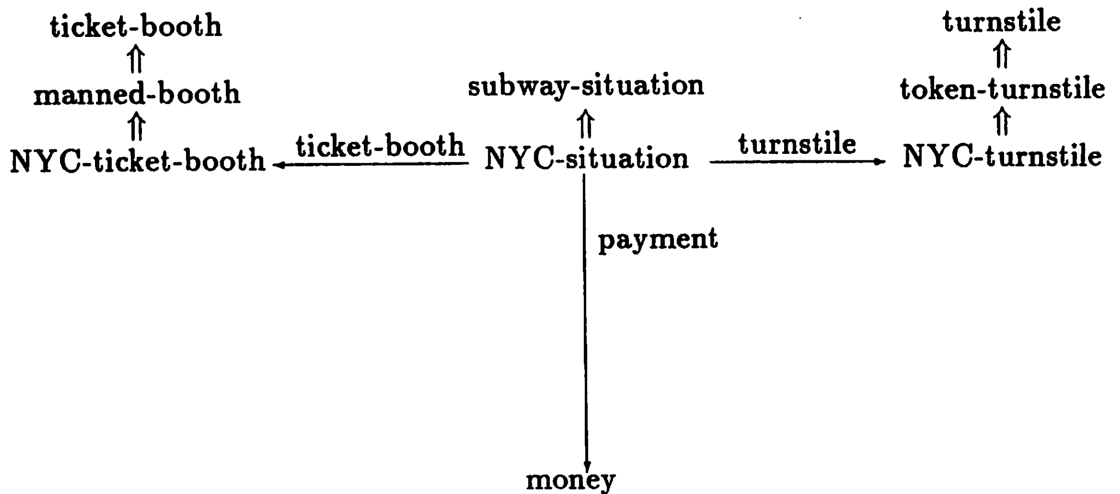

Figure 3.4: A second planning situation.

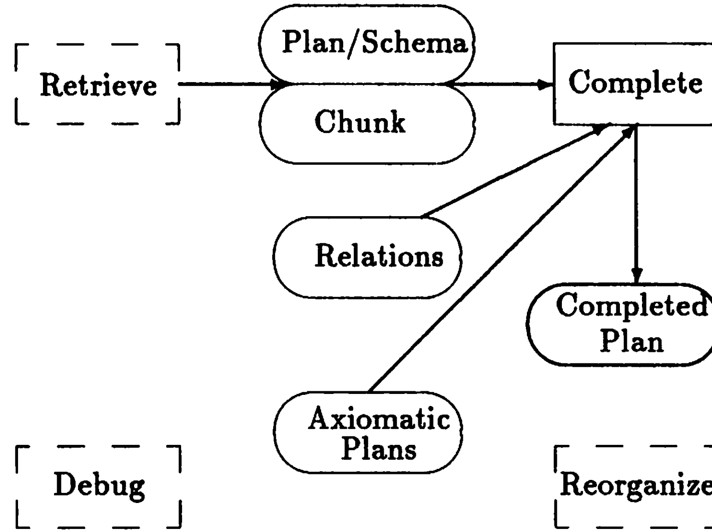

Figure 3.5: RECODER's completion component.

3.4.2 *Completion*

The completion component of the RECODER model is responsible for detailed analysis of a case. Related techniques for adapting past solutions to current problems are described in [Carbonell, 1983, Carbonell, 1986, Shinn, 1988, Alterman, 1988, Hammond, 1989]. RECODER's analysis consists of discovering inconsistencies between the current problem and the case and modifying the case to account for the inconsistencies. Prior knowledge is used extensively during completion. Hierarchical relations are used to determine inconsistencies between states and goals. Axiomatic plans are used in deriving new plan steps to modify the original case, as are previously derived chunks. Figure 3.5 details the knowledge used in completion.

Completion proceeds as follows:

- The state and goal information of a retrieved plan are compared to the current problem, and all inconsistencies between the two are noted.
- The plan is examined one step at a time. If state information for any step in the plan contains any of the aforementioned inconsistencies, the step is replaced with a sequence of steps derived from the relevant new state information, where possible.
- A retrieved plan that can be successfully completed is said to be *useful.* Useful plans are generalized with plans as they existed prior to completion, resulting in plan *schemas.*

In completion, the derivation of new plan steps from axiomatic plans is closely related to planning of the sort done by STRIPS-style planners [Fikes and Nilsson,

1971]. The planning problem is simplified in two ways:

- Whenever a sequence of steps is derived, it is compiled into a *chunk*. This chunk can then be used in similar situations in the future.
- There is no planning from scratch — planning is always done as part of the completion process. This has two benefits. First, not all plan steps need to be replaced, only those that contain inconsistencies. Second, the structure of a new plan does not need to be derived, only some of the steps within that plan.

Let us again turn to the subway domain for illustration. Suppose we are faced with the situation of being in a New York subway station (as shown in figure 3.4) and wish to complete a plan for riding the BART subway to fit that situation. The first step is to determine the inconsistencies between the two situations. There are two such inconsistencies:

1. The BART situation contains a `BART-machine`, while the New York situation contains an `NYC-ticket-booth`.
2. The BART situation contains a `BART-turnstile`, while the New York situation contains an `NYC-turnstile`.

That is, a typical BART station has a machine that dispenses farecards, while a New York station has a booth with people inside. Also, the turnstiles in the BART station are different from those in a New York station. A turnstile in a BART station returns the farecard upon entering, and the farecard is needed to exit the turnstile at the destination station. A turnstile in a New York station does not return the token given it, and no token is needed to leave the destination station.

Once these differences have been determined, we go through the BART plan one step at a time, replanning when a step contains an inconsistency. As it turns out, three of the steps in the BART plan contain inconsistencies, and the axiomatic plans to which those steps refer must be replaced with axiomatic plans that account for the inconsistencies. Below is the original BART plan, annotated with inconsistencies that arise at each step and modifications that need to be made, when necessary.

1 Buy a BART ticket at the ticket machine (some form of payment is needed for this).

 Inconsistency: *the ticket machine.*

 Modification: *replace this step with an axiomatic plan to buy a ticket from a manned ticket booth.*

2 Go through the turnstile at the station, using the ticket (the turnstile returns the ticket).

Inconsistency: *the turnstile.*

Modification: *replace this step with an axiomatic plan to pass through a turnstile that does not return the token given it.*

3 Ride the train.

Inconsistency: *none; no replanning needs to be done at this step.*

4 Exit through the turnstile (using the ticket) at the destination station.

Inconsistency: *the turnstile.*

Modification: *replace this step with an axiomatic plan to exit through a turnstile that does not require a ticket be given it.*

The completed plan for riding the New York subway is:

1 Buy a token at the manned ticket booth (some form of payment is needed for this).

2 Go through the turnstile at the station, using the token.

3 Ride the train.

4 Exit through the turnstile at the destination station.

3.4.3 Debugging

The debugging component is responsible for diagnosing and repairing failed plans. In RECODER, buggy plans are discovered with the aid of *difference maps.* A difference map is a collection of pairs of *bugs* and *repairs.*

- A *bug* is a step description in a failed plan.
- A *repair* is a mapping from an incorrect plan step to a correct plan step.

The knowledge used in debugging is shown in figure 3.6.

Continuing with our subway domain, debugging would be necessary if, e.g., there were no axiomatic plans available to complete the BART subway plan. In this case, RECODER would proceed by attempting to execute the plan until a failure occurred, and recovering from the failure by asking for advice. Below is the original plan, annotated with information about bugs and repairs.

1 Buy a BART ticket at the ticket machine (some form of payment is needed for this).

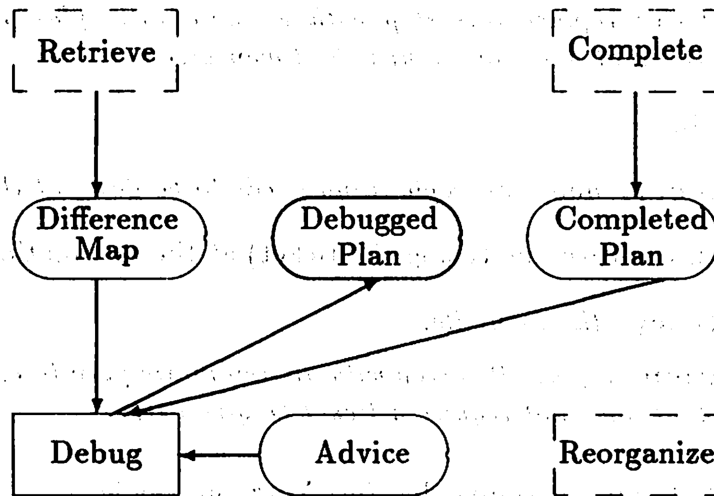

Figure 3.6: RECODER's debugging component.

Bug: ***there is no ticket machine.***

Repair: ***find a manned booth, and buy a token there.***

2 Go through the turnstile at the station, using the ticket (the turnstile returns the ticket).

Bug: ***the turnstile does not return the token.***

Repair: ***skip the part about getting anything back.***

3 Ride the train.

4 Exit through the turnstile (using the ticket) at the destination station.

Bug: ***there is no ticket to put in the turnstile.***

Repair: ***use the turnstile anyway; a ticket is not needed.***

It should be noted that the end result of debugging is the same as that of completion; in both cases, the old plan ends up being modified in the same way. There are two major differences between the methods:

1. In completion, the plan is not actually run (except in simulation); in debugging, the old plan is run, and is debugged in response to failures.

2. Completion and debugging use different types of knowledge; in debugging, the knowledge does not necessarily reside within the planner, but may exist only as outside advice.

3.4.4 Reorganization

Reorganization in RECODER includes adding new cases to memory and generalizing cases. Cases are generalized in three ways:

- When completion yields a useful plan, a generalization is made of the new plan and the plan used in the completion process. This is done by abstracting out sequences of steps that differ between the two plans, leaving enough information for the steps to be completed in the future. The result is a plan *schema.*
- When completion generates new plan steps, they are compiled into *chunks* with some constants generalized into variables. More details on chunks will be given in the next chapter.
- When debugging occurs, a difference map is constructed, as described above. Since a difference map is a description of a class of plans that will fail in a particular way, it constitutes a generalization over those plans.

Each type of generalization described above produces two pieces of information:

1. A generalized case, i.e., the schema, chunk, or difference map itself.
2. A generalized case *description*, used to index the case in memory. Case descriptions are derived from state/goal information for cases being generalized.

The reorganization component makes use of the original retrieved plan, the newly created plan, and hierarchical relations, depending on the type of generalization involved and the piece of information produced. Figure 3.7 illustrates the interaction of knowledge with the reorganization component.

We give an example of a schema using, once again, the subway domain. From the completion session given in section 3.4.2 would arise the following plan schema:[5]

1 Do something to get a token or ticket at the appropriate place (some form of payment is needed for this).

2 Do something to enter the turnstile at the station, using the token or ticket.

3 Ride the train.

4 Do something to exit via the turnstile at the destination station.

This schema will be indexed by the generalized situation shown in figure 3.8. Concepts in that figure followed by numbers (e.g., `turnstile-3`) indicate instances of the named concept.

---

[5]A plan step beginning with "Do something" indicates that no specific axiomatic plan is referenced in that step.

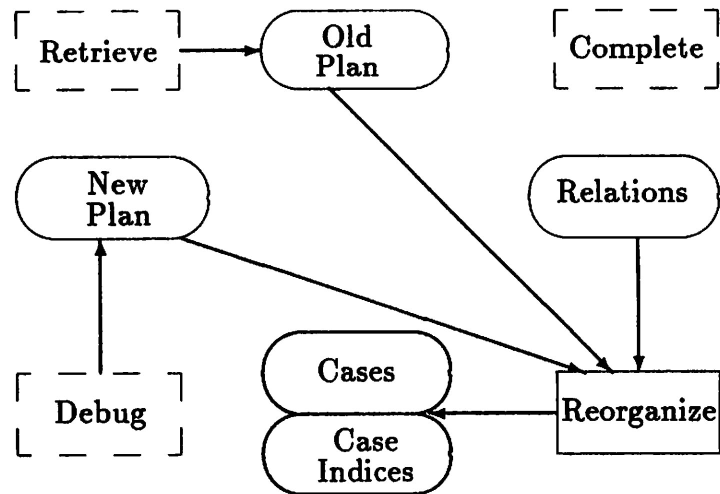

Figure 3.7: RECODER's reorganization component.

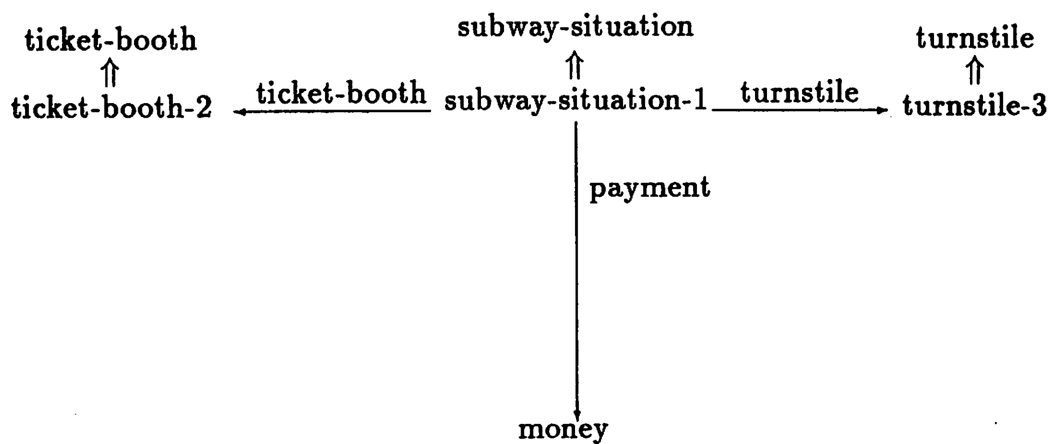

Figure 3.8: A generalized planning situation.

We turn for the last time to the subway domain to give an example of a difference map, arising from the debugging session in section 3.4.3. In this difference map, certain buggy plan steps map to empty sequences of steps. Such empty sequences are signified by the symbol ϵ.

step 1: ticket from machine $\rightarrow$ token from booth

step 2: get ticket from turnstile $\rightarrow \epsilon$

step 4: put ticket in turnstile $\rightarrow \epsilon$

This will be indexed by the differences between the two plan situations, i.e., the inconsistencies that were uncovered during completion (in section 3.4.2).

Chapter 4

The TA System

4.1 Introduction

This chapter is an in-depth presentation of TA, an implementation of the RECODER model in the domain of writing program in the LISP language [Friedman, 1974]. This introduction contains concepts that are fundamental to arguments presented in later chapters. The remainder of the chapter contains the technical details of TA, and can be skimmed on first reading.

TA is, as an implementation of RECODER, a planning system. Since TA's domain is programming, its state and goal information deal with programming concepts. For a typical TA planning session, the state information is a specification for a program and the goal is a LISP program that satisfies the specification. An example specification is: "write a program LAT? that tests whether every element of a list is an atom." An example plan to solve this problem is:

1 Construct an empty `COND` statement

2 Add the question: `(null list)`

3 Add the action: `t`

4 Add the question: `(atom (car list))`

5 Add the action: `(lat?  (cdr list))`

6 Add the question: `t`

7 Add the action: `nil`

The above plan constructs the program:

```
(cond ((null list) t)
      ((atom (car list)) (lat?  (cdr list)))
      (t nil))
```

As it turns out, both the above specification and the above plan are paraphrases of what TA actually sees. The actual form for specifications is given in section 4.2.1; the form for plans is given in section 4.2.2. In the body of the thesis, specifications will be represented by English paraphrases and plans will be represented by the programs generated by those plans, unless the details of the actual representations are necessary to the discussion at hand. Sometimes, expressions in programs will be annotated with relations to concepts in the specification. For example, the following is an annotated version of the above program, showing that the function **atom** fills the role of test in the specification (the specification concepts **function**, **spec**, and *test* will be described in section 4.2.1):

```
(cond ((null list) t)
      ((function→spec→test:atom (car list))
         (lat?  (cdr list)))
      (t nil))
```

It is important to keep in mind that, even though programming plans will generally be represented by programs rather than by plans, there is more to a programming plan than the program it generates. Even the above paraphrased plan hides some details of plans as actually represented in TA. In TA, all planning operators operate on and produce states in a state space, much as is done in planners like STRIPS [Fikes and Nilsson, 1971]. The initial specification and the goal are likewise expressed as states; the planning process is one of applying operators to states, starting with the initial state (the specification), and continuing until a goal state has been reached (signifying that a program has been constructed). Actual LISP code is constructed as a side effect of the planning process. For instance, the plan to generate an empty COND statement results in a state signifying that the statement has been generated, and as a side effect actually generates the code.[1] Planning in TA is described in more detail in sections 4.2.2 and 4.2.3.

As an automatic programming system, TA is distinctive mainly in that it takes advantage of its own programming experience in writing new programs.[2] TA shares some ideas with several popular approaches to automatic programming. Our representation for specifications is similar to that used in deductive synthesis [Manna and Waldinger, 1981]. Axiomatic plans bear a resemblance to transformation rules, as used in transformational implementation [Barstow, 1979]. The idea of encoding programming knowledge as plans is shared with the Programmer's Apprentice work

[1]This is reminiscent of the situation existing in typical robot planners, where actually carrying out an operation (such as moving a block) is distinct from applying an analogous operation to a state in a state space.

[2]TA is not completely alone in using past programs to aid in current programming problems; [Dershowitz, 1986] is another example of work in this area. Similarly, learning programming concepts has been addressed elsewhere, notably [Anderson, 1986] and [Burstein, 1986].

[Rich, 1981, Waters, 1985], as is the notion of plan schemas (though they refer to them as "cliches").

A typical program writing episode in TA follows the RECODER model according to the following algorithm:

1. *Retrieval*: TA attempts to retrieve a plan schema from the schema case base. If no applicable schema is found, TA retrieves a plan to construct a program from the plan case base.

2. *Completion*: If a plan was retrieved, TA generates new steps to replace any inconsistent steps in the retrieved plan. New steps are generated either by applying chunks (if any are applicable) or by applying axiomatic plans. Plan schemas are instantiated by filling in empty slots with newly generated plan steps. If a plan was successfully completed, a plan schema is created.

3. *Debugging*: The plan generated by completion is applied to yield a program. TA receives (from the user) a sample call of the program being written. If the call is unsuccessful, difference maps are searched for and, if found, applied. If necessary, new difference maps are created with user direction. The program is modified as necessary.

4. *Reorganization*: When a program has been determined to be correct, cases are added to the appropriate case bases.

4.2 Knowledge Sources in TA

This section gives, in detail, the prior knowledge available to TA in the form of hierarchical relations and axiomatic plans, and shows an example of plan application. Knowledge in the form of advice is given either as a response to a "yes/no" question or as a LISP expression.

4.2.1 Hierarchical Relations

Figures 4.1 through 4.9 show the concepts in TA's programming domain.[3]

The high-level concepts available to TA are shown in figure 4.1. We can see that for TA, a function consists of three parts: a specification (or `spec`) that describes what the function is to do, one or more `inputs` to the function, and `code` that

[3]In many of these figures, a group of concepts is enclosed in an oval, with the oval taking part in one or more relations. This is to be interpreted as all of the concepts within the oval taking part in the relations.

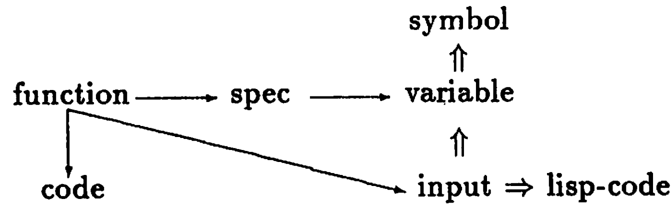

Figure 4.1: High-level concepts about programs in TA.

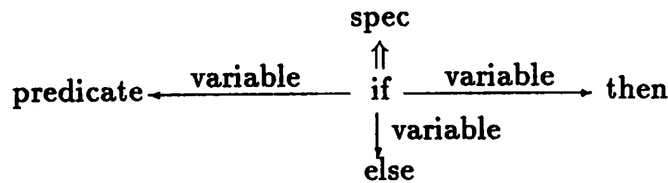

Figure 4.2: The `if` specification concept.

implements the specification. A `spec` contains one or more `variables`. Each `input` to the function is `lisp-code`, and is also a `variable`. All `variables` are `symbols`.

Concepts in TA's specification language are shown in figures 4.2, 4.3, and 4.4. Figure 4.2 shows the `if` specification concept. This concept contains three variables: `predicate`, `then`, and `else`.

Figure 4.3 shows several specification concepts. Each of these concepts has a `variable` that is a `collection`. The top subgroup shows concepts that iterate over items in a collection; each of these contains two additional parts called a `test` and an `element`. This group contains a further subgroup whose concepts respond to some `message` when a test is true for an element.

The second subgroup in figure 4.3 shows concepts that contain a second variable called an `object`. The third subgroup shows concepts that have two additional variables, a `target` and a `new-object`.

Figure 4.4 shows a group of specification concepts each of which refers to its `variable` as an `argument`. There are two concepts in the figure, `same` and `included-in`, that have a second variable called a `comparator`.

A specification as presented to TA looks somewhat like an expression in logic. For example, a program that tests whether all elements of its single input `list` are atoms would be presented to TA as `(for-all x list (atomic x))`. TA translates this into the network of concepts shown in figure 4.5. The translation of an expression (x y_1 ... y_n) is in two parts:

1. TA makes an instance of the concept x named in the first position of the expression. In the example above, this results in `for-all117`, an instance of `for-all`, being created.

2. TA determines the parts of the instance it just created by examining the order

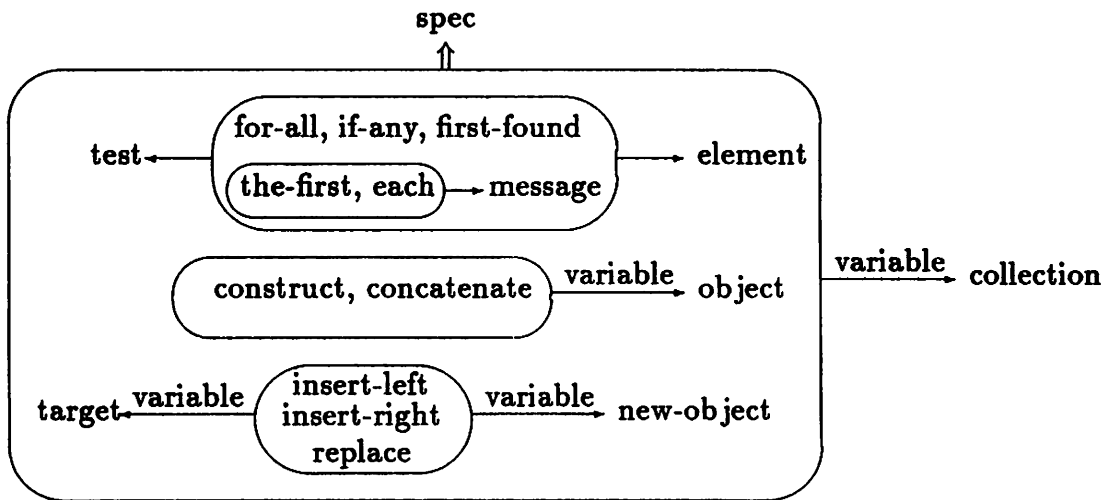

Figure 4.3: Specification concepts dealing with collections.

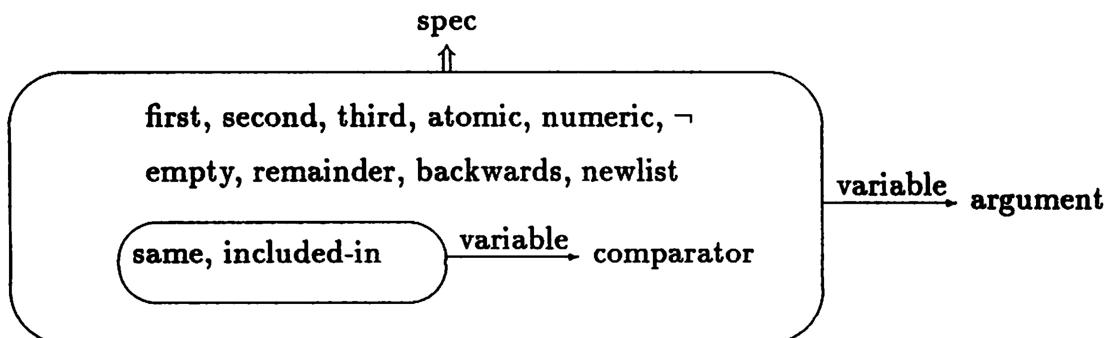

Figure 4.4: More specification concepts.

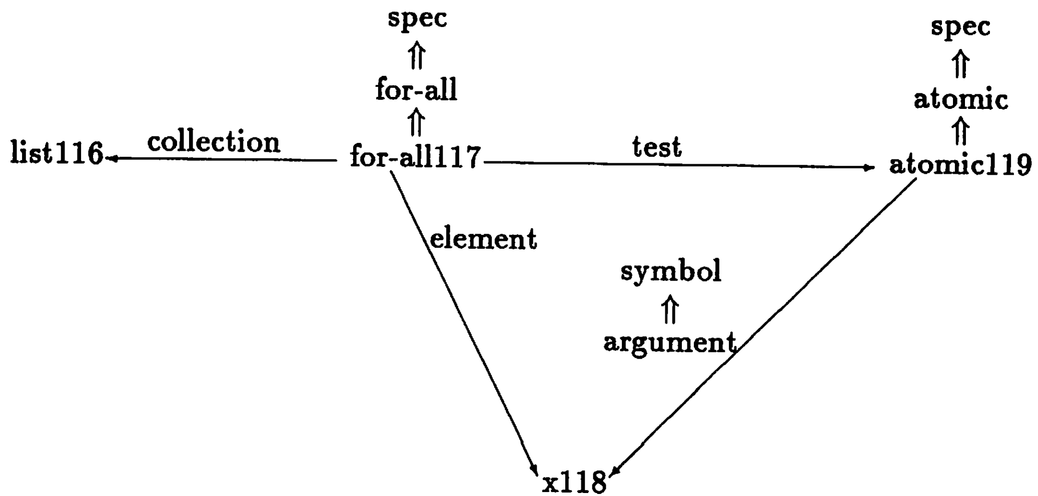

Figure 4.5: A specification represented as a network of concepts in TA.

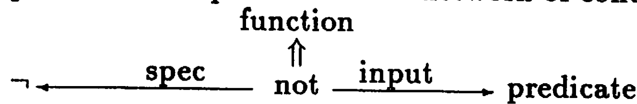

Figure 4.6: The **not** program concept.

of the arguments y_1 ... y_n of the expression. For the **for-all** concept, TA knows that the first argument is the **element** of the concept, the second the **collection** of the concept, and the third the **test** of the concept. Instances of these concepts are generated. If any argument is itself an expression, TA recursively translates the argument. In the example above, this occurs for the subexpression **atomic x**.

In figures 4.6, 4.7, 4.8, and 4.9, concepts about specifications are related to concepts in the LISP programming language. Figure 4.6 shows the **not** program concept, which relates to the ¬ specification concept and which takes a **predicate** as its input.

Figure 4.7 shows several program concepts, each of which takes a single **expression** as input.

Program concepts that take two inputs are shown in figure 4.8. All of them have an **object-expression** input; one subgroup has an additional **comparator-expression** input and another has an additional **collection-expression** input.

Figure 4.9 shows two program concepts that take three inputs: a **collection-expression**, a **target-expression**, and a **new-object-expression**.

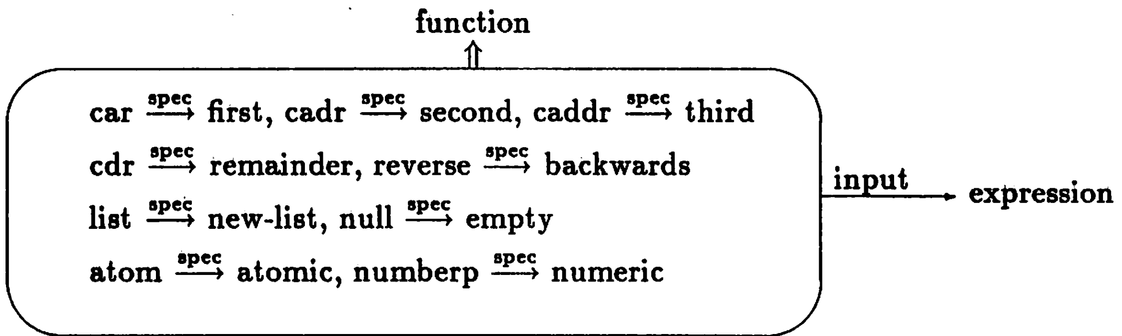

Figure 4.7: Single input program concepts.

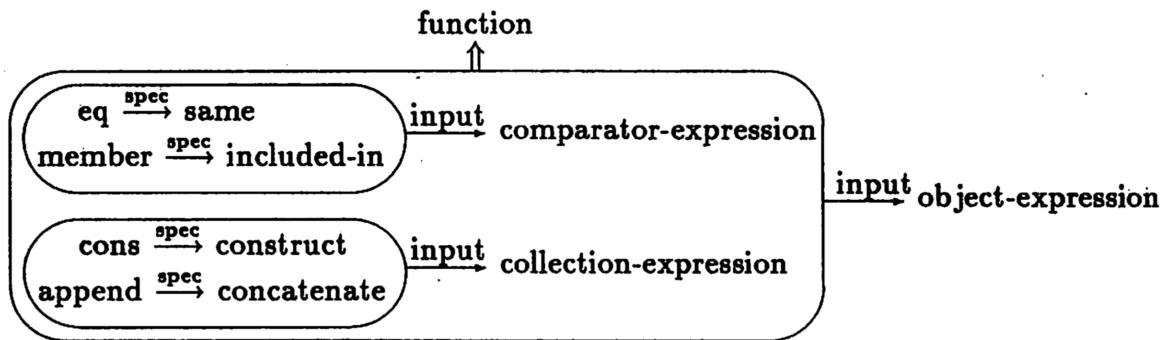

Figure 4.8: Two-input program concepts.

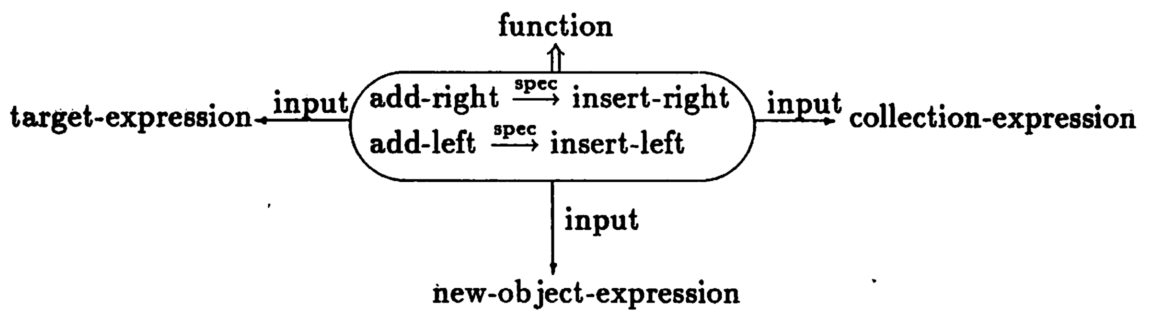

Figure 4.9: Three-input program concepts.

4.2.2 Axiomatic Plans

Earlier, the following plan was presented as an example of the kind plan used in writing programs:

1 Construct an empty `COND` statement

2 Add the question: `(null list)`

3 Add the action: `t`

4 Add the question: `(atom (car list))`

5 Add the action: `(lat?  (cdr list))`

6 Add the question: `t`

7 Add the action: `nil`

This plan was suggested as a plan to generate the program:

```
(cond ((null list) t)
      ((atom (car list)) (lat? (cdr list)))
      (t nil))
```

In fact, the plan as represented by TA is somewhat more involved than the paraphrase above. The actual plan is:

```
0 (GENERATE-SYMBOL CONDITIONAL1)

1 (GENERATE-COND CONDITIONAL1 (CODE FUNCTION))

    1.1 (GENERATE-QUESTION CONDITIONAL1 QUESTION1)
    1.2 (GENERATE-ACTION CONDITIONAL1 ACTION1)

2 (->NULL QUESTION1 (COLLECTION (SPEC FUNCTION)))

3 (->T ACTION1)

    3.1 (GENERATE-QUESTION CONDITIONAL1 QUESTION2)
    3.2 (GENERATE-ACTION CONDITIONAL1 ACTION2)
    3.3 (->CAR (ELEMENT (SPEC FUNCTION)) (COLLECTION (SPEC FUNCTION)))

4 (->ATOM QUESTION2 (ELEMENT (SPEC FUNCTION)) (TEST (SPEC FUNCTION)))

    4.1 (->CDR SYMBOL1 (FIRST-ARGUMENT FUNCTION))
```

```
5 (->RECURSE1 ACTION2 FUNCTION SYMBOL1)
    5.1 (GENERATE-QUESTION CONDITIONAL1 QUESTION3)
    5.2 (GENERATE-ACTION CONDITIONAL1 ACTION3)
6 (->T QUESTION3)
7 (->NIL ACTION3)
8 (GENERATE-LISP-CODE FUNCTION)
```

Each step in this more detailed plan denotes a further plan. Plans thus denoted can be *primitive* plans, which are not decomposable, or they can be *non-primitive* plans, which are decomposed into further plans. All primitive plans are axiomatic; non-primitive plans may or may not be. In TA, we use the convention that non-primitive plans begin with the characters `->`, and primitive plans do not. Only non-primitive plans are available for reasoning about during planning.[4]

In the planning model employed by TA, the planning process consists of augmenting the hierarchical relations between concepts available to TA. For example, the goal of a plan that generates Lisp code for a function f is satisfied if $f \Rightarrow$ `lisp-code`. Many of the primitive plans available to TA result in the creation of such a hierarchical relation. However, merely creating such relations will not produce LISP programs, just as stating that several blocks have been stacked will not in itself cause those blocks to be stacked in a typical robot planner. Such actions take place as a side effect of the symbolic process that is planning. Thus, many of the plans in TA that create hierarchical relations also cause fragments of LISP code to be written as side effects. Each such fragment is associated with a concept available to the plan. The code fragment of a concept c is denoted $\Box c$.[5]In addition to fragments of code, a concept may have a list of concepts associated with it. The list associated with a concept c is denoted $\ell[c]$.

There are nine primitive plans in TA that result in new relations between concepts. Each plan is listed below with its associated state, the results of applying the plan, and any side effects caused by the plan.

1. `generate-symbol`.
   state: A concept c.

[4]Primitive and non-primitive plans may be distinguished by analogy to a human being. Primitive plans correspond to actions that are driven purely by motor impulse (e.g., "lift arm"), while non-primitive plans organize other plans towards achieving a goal (e.g., "answer the telephone"). Non-primitive plans may be reasoned about by virtue of their being goal-directed; primitive plans, not being goal-directed, are not available for reasoning.

[5]Note, however, that there are some concepts c for which $\Box c \equiv c$, e.g., concepts representing LISP symbols.

result: $c \Rightarrow$ `symbol`.
side effects: none.

2. `generate-cond`.
   state: A concept c, a concept s.
   result: $s \Rightarrow$ `symbol`; $c \Rightarrow$ `conditional`
   side effects: $\Box c$ becomes `(COND)`, $\Box s$ becomes c.

3. `generate-question`.
   state: A `conditional` c, a concept s.
   result: $s \Rightarrow$ `symbol`.
   side effects: s is inserted in $\Box c$ as a new `COND` question; i.e., if $\Box c$ is `(COND` `(`x y`))` before this plan is applied, the plan will cause $\Box c$ to become `(COND` `(`x y`)` `(`s`))`.

4. `generate-action`.
   state: A `conditional` c, a concept s.
   result: $s \Rightarrow$ `symbol`.
   side effects: s is inserted in $\Box c$ as a new `COND` action; i.e., if $\Box c$ is `(COND` `(`x y`)` `(`z`))` before this plan is applied, the plan will cause $\Box c$ to become `(COND` `(`x y`)` `(`z s`))`.

5. `generate-constant`.
   state: A `symbol` s, a constant c.
   result: $s \Rightarrow$ `lisp-code`.
   side effects: $\Box s$ becomes c.

6. `generate-unary-function`.
   state: A `symbol` s, a function name f, an argument a.
   result: $s \Rightarrow$ `lisp-code`.
   side effects: $\Box s$ becomes $(f\ a)$.

7. `generate-n-ary-function`.
   state: A `symbol` s, a function name f, a concept a with an associated argument list.
   result: $s \Rightarrow$ `lisp-code`.
   side effects: $\Box s$ becomes $(f\ .\ \ell[a])$.[6]

8. `retrieve-schema`.
   state: A `spec` s, a symbol f.
   result: $f \Rightarrow$ `function`; in addition, all concepts x such that $s \Rightarrow x$ or $s \rightarrow x$ are copied, and the copies are given analogous relations to f.

[6] This is written in dotted list notation, e.g., $(x\ .\ (y\ z)) = (x\ y\ z)$.

side effects: A case matching s is retrieved from the schema case base and is associated with f.

9. `generate-lisp-code`.
   state: A `function` f, the code c of f.
   result: A new symbol s such that $s \Rightarrow$ `lisp-code`.
   side effects: $\Box s$ becomes the *LISP expansion* of c, defined below.

The LISP expansion of an expression e, denoted $\mathcal{L}(e)$, is defined recursively as follows:

- If e is atomic and $\Box e$ is empty, $\mathcal{L}(e) = e$.
- If e is atomic and $\Box e$ is not empty, $\mathcal{L}(e) = \mathcal{L}(\Box e)$.
- If e is not atomic, then there exists an x and a y such that $e = (x\ .\ y)$, and $\mathcal{L}(e) = (\mathcal{L}(x)\ .\ \mathcal{L}(y))$.

There are three additional primitive plans which are used for side effects only. They are:

1. `associate`.
   state: A `symbol` s, a concept v.
   side effects: v is appended to the end of $\ell[s]$.

2. `apply-schema`.
   state: A schema s.
   side effects: s is completed.

3. `generate-call`n.
   state: A `function` f, a `symbol` s, n arguments $a_1 ... a_n$.
   side effects: $\Box f$ becomes $(s\ a_1\ ...\ a_n)$.

In addition to the primitive plans, TA is provided with 21 non-primitive plans. With each plan is listed its state and goal information, as well as the steps that compose the plan. State and goal information is paraphrased; for instance, the phrase "lisp code l that is the argument of c" would be more precisely rendered as "a concept l such that $l \Rightarrow$ `lisp-code` and $c \xrightarrow{\text{argument}} l$." The goal of each plan is "lisp code generated for s," where s is a state item such that $s \Rightarrow$ `symbol`. More precisely, this means that after the plan is executed, s should be an instance of `lisp-code`; that is, it should be the case that $s \Rightarrow$ `lisp-code`.

1. `->t`.
   state: a symbol s.
   goal: lisp code generated for s.
   steps: (generate-constant s 't)

2. `->nil`.
state: a symbol s.
goal: lisp code generated for s.
steps: (generate-constant s 'nil)

3. `->constant`.
state: a symbol s, some lisp code l.
goal: lisp code generated for s.
steps: (generate-constant s l)

4. `->car`.
state: an occurrence c of the concept `first`, a symbol s, lisp code l that is the argument of c.
goal: lisp code generated for s.
steps: (generate-unary-function s 'car l)

5. `->cadr`.
state: an occurrence c of the concept `second`, a symbol s, lisp code l that is the argument of c.
goal: lisp code generated for s.
steps: (generate-unary-function s 'cadr l)

6. `->caddr`.
state: an occurrence c of the concept `third`, a symbol s, lisp code l that is the argument of c.
goal: lisp code generated for s.
steps: (generate-unary-function s 'caddr l)

7. `->cdr`.
state: an occurrence c of the concept `remainder`, a symbol s, lisp code l that is the argument of c.
goal: lisp code generated for s.
steps: (generate-unary-function s 'cdr l)

8. `->reverse`.
state: an occurrence c of the concept `backwards`, a symbol s, lisp code l that is the argument of c.
goal: lisp code generated for s.
steps: (generate-unary-function s 'reverse l)

9. `->list`.
state: an occurrence c of the concept `new-list`, a symbol s, lisp code l that is the argument of c.
goal: lisp code generated for s.
steps: (generate-unary-function s 'list l)

10. `->null`.
state: an occurrence c of the concept `empty`, a symbol s, lisp code l that is the argument of c.
goal: lisp code generated for s.
steps: (generate-unary-function s 'null l)

11. `->atom`.
state: an occurrence c of the concept `atomic`, a symbol s, lisp code l that is the argument of c.
goal: lisp code generated for s.
steps: (generate-unary-function s 'atom l)

12. `->numberp`.
state: an occurrence c of the concept `numeric`, a symbol s, lisp code l that is the argument of c.
goal: lisp code generated for s.
steps: (generate-unary-function s 'numberp l)

13. `->eq`.
state: an occurrence c of the concept `same`, a symbol s, lisp code x that is the argument of c, lisp code y that is the comparator of c.
goal: lisp code generated for s.
steps:
(generate-symbol args)
(associate args x)
(associate args y)
(generate-n-ary-function s 'eq args)

14. `->member`.
state: an occurrence c of the concept `included-in`, a symbol s, lisp code x that is the argument of c, lisp code y that is the comparator of c.
goal: lisp code generated for s.
steps:
(generate-symbol args)
(associate args y)
(associate args x)
(generate-n-ary-function s 'member args)

15. `->not`.
state: an occurrence c of the concept $\neg$, a symbol s, lisp code l that is the argument of c.
goal: lisp code generated for s.
steps: (generate-unary-function s 'not l)

16. `->cons`.
state: an occurrence c of the concept `construct`, a symbol s, lisp code x that is the object of c, lisp code y that is the collection for c.
goal: lisp code generated for s.
steps:
(generate-symbol args)
(associate args x)
(associate args y)
(generate-n-ary-function s 'cons args)

17. `->replace`.
state: an occurrence c of the concept `replace`, a symbol s, lisp code x that is the new object for c, lisp code y that is the target for c, lisp code z that is the collection for c.
goal: lisp code generated for s.
steps:
(generate-symbol args)
(associate args x)
(associate args z)
(generate-n-ary-function s 'cons args)

18. `->append`.
state: an occurrence c of the concept `concatenate`, a symbol s, lisp code x that is the object of c, lisp code y that is the collection for c.
goal: lisp code generated for s.
steps:
(generate-symbol args)
(associate args x)
(associate args y)
(generate-n-ary-function s 'append args)

19. `->add-right`.
state: an occurrence c of the concept `insert-right`, a symbol s, lisp code x that is the new object for c, lisp code y that is the target for c, lisp code z that is the collection for c.
goal: lisp code generated for s.
steps:
(generate-symbol args)
(associate args x)
(associate args y)
(associate args z)
(generate-n-ary-function s 'add-right args)

20. `->add-left`.
state: an occurrence c of the concept `insert-left`, a symbol s, lisp code x that is the new object for c, lisp code y that is the target for c, lisp code z that is the collection for c.
goal: lisp code generated for s.
steps:
(generate-symbol args)
(associate args x)
(associate args y)
(associate args z)
(generate-n-ary-function s 'add-left args)

21. `->recursen`.
state: a function f, a symbol s, n pieces of lisp code x_1 ... x_n.
goal: lisp code generated for s.
steps:
(generate-symbol args)
for each $i, 1 \leq i \leq n$: (associate args x_i)
(generate-n-ary-function s f args)

In general, there will be one non-primitive plan for every expression that TA is able to generate. These plans thus correspond to TA's facility with concepts in the LISP programming language. Obviously, a set of only 21 non-primitive plans implies that TA can only take advantage of a small subset of LISP. These plans are relatively modular, however, and new non-primitive plans may be added as greater flexibility is desired.

On the other hand, the set of primitive plans is intended to be fixed, and corresponds more to intuitions about writing programs down on paper than to facility with manipulating concepts in the programming language. These plans deal in general with the syntactic form the final program will take.

4.2.3 An Example of Plan Application

To illustrate how TA constructs programs, we present a plan that is annotated at each step with an English paraphrase of the step and the effects of applying that step. This plan is for the program `LAT?`, whose specification was shown in figure 4.5. The planning state is the collection of all concepts appearing in the specification, together with those concepts added during the planning process. After all steps of the plan have been applied, the definition of `LAT?` is associated with the concept referred to in the plan as `LISP-CODE1`.

`0 (GENERATE-SYMBOL CONDITIONAL1)`
paraphrase: Make `conditional1` a symbol.

effects: conditional1 ⇒ symbol.

1 (GENERATE-COND CONDITIONAL1 (CODE FUNCTION))
paraphrase: Make conditional1 a conditional; associate it with the code of the function to be generated (i.e., lat?).
effects: conditional1 ⇒ conditional; (code lat?) ⇒ symbol;
□ conditional1 = (COND); □ (code lat?) = conditional1.

1.1 (GENERATE-QUESTION CONDITIONAL1 QUESTION1)
paraphrase: Make question1 a new question for conditional1.
effects: question1 ⇒ symbol; □ conditional1 = (COND (question1)).

1.2 (GENERATE-ACTION CONDITIONAL1 ACTION1)
paraphrase: Make action1 a new action for conditional1.
effects: action1 ⇒ symbol; □ conditional1 = (COND (question1 action1)).

2 (->NULL QUESTION1 (COLLECTION (SPEC FUNCTION)))
paraphrase: Generate the LISP code (null *expression*), where *expression* is the collection of the spec of lat?. Associate this code with question1.
effects: list116 ⇒ lisp-code; □ list116 = (null list116).

3 (->T ACTION1)
paraphrase: Associate the LISP expression t with action1.
effects: action1 ⇒ lisp-code; □ action1 = t.

3.1 (GENERATE-QUESTION CONDITIONAL1 QUESTION2)
paraphrase: Make question2 a new question for conditional1.
effects: question2 ⇒ symbol;
□ conditional1 =

```
(COND (question1 action1) (question2)).
```

3.2 (GENERATE-ACTION CONDITIONAL1 ACTION2)
paraphrase: Make action2 a new action for conditional1.
effects: action2 ⇒ symbol;
□ conditional1 =

```
(COND (question1 action1)
      (question2 action2)).
```

3.3 (->CAR (ELEMENT (SPEC FUNCTION)) (COLLECTION (SPEC FUNCTION)))

paraphrase: Generate the LISP code (car *expression*), where *expression* is the collection of the spec of lat?. Associate this code with the element of the spec of lat?.
effects: x118 ⇒ lisp-code; □ x118 = (car list116).

4 (->ATOM QUESTION2 (ELEMENT (SPEC FUNCTION)) (TEST (SPEC FUNCTION)))

paraphrase: Assuming that the test of the spec of lat? is an instance of atomic, generate the LISP code (atom *expression*), where *expression* is the element of the spec of lat?. Associate this code with question2.
effects: question2 ⇒ lisp-code; □ question2 = (atom x118).

4.1 (->CDR SYMBOL1 (FIRST-ARGUMENT FUNCTION))
paraphrase: Generate the LISP code (cdr *expression*), where *expression* is the first argument of lat?. Associate this code with symbol1.
effects: symbol1 ⇒ lisp-code; □ symbol1 = (car list116).

5 (->RECURSE1 ACTION2 FUNCTION SYMBOL1)
paraphrase: Generate the LISP code (lat? symbol1) and associate it with action2.
effects: action2 ⇒ lisp-code; □ action2 = (lat? symbol1)

5.1 (GENERATE-QUESTION CONDITIONAL1 QUESTION3)
paraphrase: Make question3 a new question for conditional1.
effects: question3 ⇒ symbol;
□ conditional1 =

```
(COND (question1 action1)
      (question2 action2)
      (question3)).
```

5.2 (GENERATE-ACTION CONDITIONAL1 ACTION3)
paraphrase: Make action3 a new action for conditional1.
effects: action3 ⇒ symbol;
□ conditional1 =

```
(COND (question1 action1)
      (question2 action2)
      (question3 action3)).
```

6 (->T QUESTION3)
paraphrase: Associate the LISP expression t with question3.
effects: question3 ⇒ lisp-code; □ question3 = t.

7 (->NIL ACTION3)
paraphrase: Associate the LISP expression nil with action3.
effects: action3 ⇒ lisp-code; □ action3 = nil.

8 (GENERATE-LISP-CODE FUNCTION)
paraphrase: Generate LISP code for lat?.

effects: `lisp-code1` is created; `lisp-code1` ⇒ `lisp-code`;
□ `lisp-code1` = $\mathcal{L}$(`(code lat?)`) =

```
(COND ((null list116) t)
      ((atom (car list116)) (lat? (cdr list116)))
      (t nil)).
```

The above plan contains a total of 17 steps, 9 referring to primitive plans and 8 to non-primitives. In general, there will be one non-primitive plan for each expression in the final program. In the above plan, for example, there are 8 expressions within the COND clause (including the two nested expressions). The primitive plans are generally used to associate expressions with symbols, and thus there will also be about the same number of primitive plans as expressions in general. Thus, a plan will as a rule contain about twice as many steps as there are expressions in the generated program.

4.3 TA in Action

This section will illustrate each of the components of TA via an extended example. More details on the algorithms involved in each component appear in later chapters. Details on case retrieval are in chapter 5; details on completion and debugging are in chapter 6. Reorganization via creation of both difference maps and chunks is discussed in chapter 6. Schema creation is described in detail in chapter 7.

4.3.1 Retrieval

Retrieval in TA is begun by sending a description of the current situation to all of the cases in memory. Each case matches its own description to the description of the current situation, and a parallel voting algorithm selects the best matching case.[7]

An example from TA will help illustrate the retrieval process. Suppose that TA's plan case base contains a plan for constructing the program `LAT?`, which tests whether its single input `list` is a list of atoms. `LAT?` is specified to TA as `(for-all x list (atomic x))`. This specification is represented internally as the concept shown in figure 4.10.

Suppose now that TA is given the specification for a program `ALL-EMPTY?`, which tests whether its single input `list` is a list of empty lists. `ALL-EMPTY?` is specified as `(for-all x list (empty x))`, and is represented internally as the concept shown in figure 4.11.

[7] The parallel algorithm is described in chapter 5.

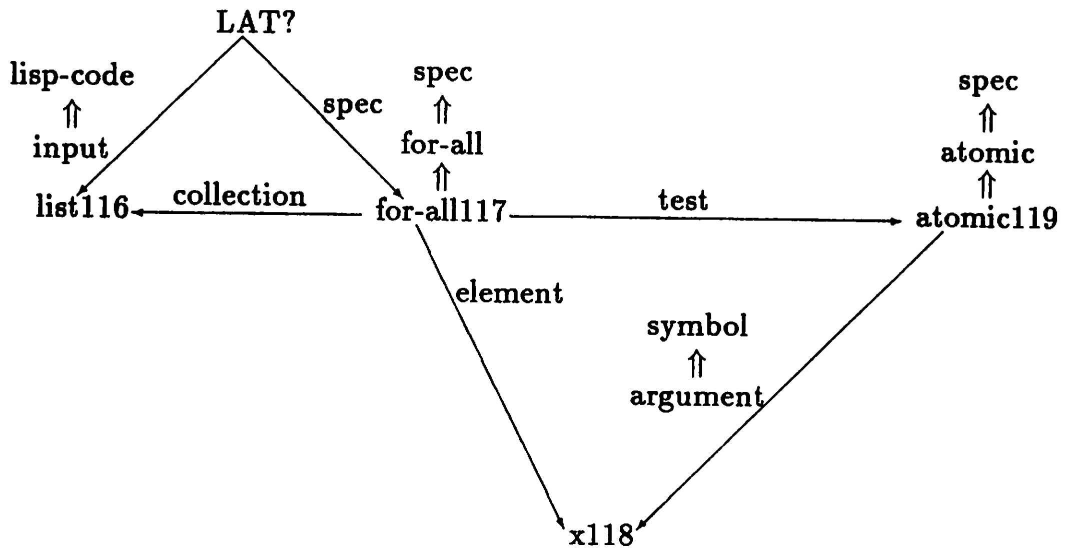

Figure 4.10: A network representation for LAT?.

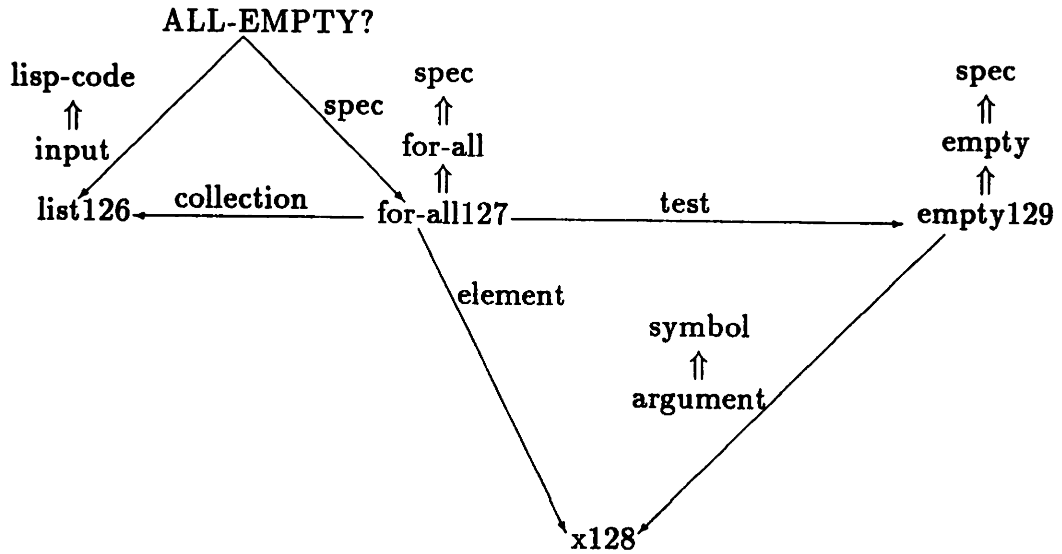

Figure 4.11: A network representation for ALL-EMPTY?.

The internal representation is treated as the description of the current situation and sent to the case base,[8] where it is compared with each case. In our example, the plan for LAT? is the only case in the case base, so it is the only one compared.

Case descriptions are compared to one another by counting the number of components that match. Components of two descriptions match if they are instances of the same parent concept. In our example, the specification for ALL-EMPTY? has four components: `for-all127`, `x128`, `list126`, and `empty129`. It is determined that all of these components match a component of the specification for LAT? except for `empty129`. This has no match since there is no concept x in LAT?'s specification such that $x \Rightarrow$ `empty`.

In general, the case with the greatest number of matching components is selected as the best case. Since the plan for LAT?, with three matching components, is the only case in the case base, it is selected by default. A case must have at least one matching component in order to be selected; if no case has at least one matching component, then no case is retrieved.

4.3.2 *Completion*

Continuing our example with TA, suppose that TA has retrieved the plan for constructing the program LAT?, given the task of writing a program ALL-EMPTY?. The code for LAT? is as follows:

```
(cond ((null list) t)
      ((atom (car list)) (lat? (cdr list)))
      (t nil))
```

The associated plan, in moderate detail, is:[9]

```
1 (->NULL QUESTION1 (COLLECTION (SPEC FUNCTION)))

2 (->T ACTION1)

3 (->CAR (ELEMENT (SPEC FUNCTION)) (COLLECTION (SPEC FUNCTION)))

4 (->ATOM QUESTION2 (ELEMENT (SPEC FUNCTION)) (TEST (SPEC FUNCTION)))

5 (->CDR SYMBOL1 (FIRST-ARGUMENT FUNCTION))
```

[8]TA in fact decomposes the description into a set of attribute/value features; it is these features that are sent to the case base. The process is described in chapter 5.

[9]By "moderate detail," we mean that all non-primitive steps are shown, but primitive steps are not shown. This level of detail is sufficient for illustrating the completion process, since only non-primitive steps are reasoned about during completion.

6 (->RECURSE1 ACTION2 FUNCTION SYMBOL1)

7 (->T QUESTION3)

8 (->NIL ACTION3)

The first step in the completion process is to find the inconsistencies between the state/goal information for the retrieved case and that for the problem to be solved. In TA, such information takes the form of program specifications.

Comparing the specifications for LAT? and ALL-EMPTY? yields the following three inconsistencies:

1. The specification for LAT? contains a concept called LIST116, while that for ALL-EMPTY? contains a concept called LIST126.

2. The specification for LAT? contains a concept called X118, while that for ALL-EMPTY? contains a concept called X128.

3. The specification for LAT? contains a concept called ATOMIC119, while that for ALL-EMPTY? contains a concept called EMPTY129.

Of these, it is determined that only the inconsistency involving ATOMIC119 and EMPTY129 is *relevant.* An inconsistency (o, n) is defined to be relevant iff either o or n can be decomposed into parts. From figures 4.10 and 4.11, we can see that EMPTY129 and ATOMIC119 can both be so decomposed, but none of the concepts involved in the other inconsistencies can.

Once inconsistencies are found, the plan for LAT? is examined a step at a time, looking for steps that refer to relevant inconsistencies. The only such step is step 4, which refers to the test of the spec of the function, i.e., ATOMIC116 in the specification for LAT? and EMPTY129 in the specification for ALL-EMPTY?. Thus, steps 1-3 and 5-8 may remain as they are for ALL-EMPTY?, but step 4 must be replaced.

TA uses the state of the to-be-discarded plan step, as it applies to the specification for ALL-EMPTY?, as an initial state in creating a sequence of replacement plan steps. The state includes the symbol QUESTION2, the element of the spec of the function (i.e., X128), and the test of the spec of the function (i.e., EMPTY129). TA discovers the axiomatic plan ->NULL, which includes in its state state: an occurrence c of the concept empty, a symbol, and lisp code that is the argument of c. Since X128 is in fact the argument of EMPTY129 (see figure 4.11), the state of this plan is a perfect match.[10] Step 4 of the plan for ALL-EMPTY? becomes:

[10]It is not necessary to find a plan with a perfectly matching state. If the best available plan has a state that only partially matches, it is assumed that more than one replacement step is needed, and planning continues.

(->NULL QUESTION2 (ELEMENT (SPEC FUNCTION)) (TEST (SPEC FUNCTION))). This step is substituted into the original plan, and completion is successful.

Since completion was successful, a new plan schema is created:

```
1 (->NULL QUESTION1 (COLLECTION (SPEC FUNCTION)))

2 (->T ACTION1)

3 (->CAR (ELEMENT (SPEC FUNCTION)) (COLLECTION (SPEC FUNCTION)))

4 (* QUESTION2 (ELEMENT (SPEC FUNCTION)) (TEST (SPEC FUNCTION)))

5 (->CDR SYMBOL1 (FIRST-ARGUMENT FUNCTION))

6 (->RECURSE1 ACTION2 FUNCTION SYMBOL1)

7 (->T QUESTION3)

8 (->NIL ACTION3)
```

This plan schema can be viewed as generating, not a program, but a *program schema*. The program schema generated in this case is:

```
(cond ((null list) t)
      ((function→spec→test (car list))
        (function (cdr list)))
      (t nil))
```

This differs from the original program in that two concepts, the **test** of the **spec** of the **function** and the **function** itself, are not given explicitly.

4.3.3 Debugging

The debugging component of TA first checks to see if the plan being constructed has been completed successfully. If so, TA simply returns the completed plan. If not, TA looks for appropriate difference maps, and applies them if found. TA then enters an interactive debugging session which continues until the advice TA receives indicates that the program has been successfully debugged.

To illustrate the debugging process, suppose that TA is asked to write HAS-ATOM?, which is to test whether its single input l contains an atom. This is specified to TA as (if-any x l (atomic x)). Suppose further that the best match in TA's plan case base is the plan for LAT?. The specifications for LAT? and HAS-ATOM? have a single relevant inconsistency: LAT? contains an instance of for-all, while HAS-ATOM? contains an instance of if-any. Unfortunately, no step in the plan for

writing **LAT?** refers directly to its instance of `for-all`; and even if there were such a step, it could not be replaced, since there are no axiomatic plans dealing with `if-any`. The result is that TA cannot successfully complete the plan, and so it enters debugging phase with the program for **LAT?** as its first approximation.

Let us assume that no difference maps are yet available; TA must examine traces of program calls to find bugs and repairs.

The first call that is traced is (HAS-ATOM? '()). TA's program returns **T** for this, but discovers through asking for advice that the correct answer is **NIL**. TA looks at the first **COND** question, which asks `(null '())`. TA's program returns **T** for this expression, which agrees with the advice it receives. TA then looks at the associated **COND** action, which is the expression **T**. It receives advice telling it that **NIL** should be returned, and that the correct code is the expression **NIL**.

Next, TA traces a call to (HAS-ATOM? '(a (x))). Tracing the program reveals that the action of the second **COND** clause is incorrect; the correct action should be the expression **T**.

Finally, a call to (HAS-ATOM? '((x) a)) reveals that the action of the third **COND** clause is incorrect; it should contain a recursive call to **HAS-ATOM?**.

With the above three repairs, the program for **HAS-ATOM?** is deemed correct. The following difference map is created, which summarizes the results of the debugging session (italicized items denote variables):

action 1: `t` → `nil`

action 2: `(`*function* `(cdr` *variable*`))` → `t`

action 3: `nil` → `(`*function* `(cdr` *variable*`))`

Application of the difference map results in the following program:

```
(cond ((null list) nil)
      ((atom (car list)) t)
      (t (has-atom?  (cdr list))))
```

The difference map and the plan corresponding to the above program are stored in the appropriate case bases.

4.3.4 *Reorganization: Creating Difference Maps*

The following example will illustrate the process of creating difference maps and their descriptions in TA.

Consider the program **HAS-ATOM?** as derived from **LAT?** in the example in section 4.3.3. We saw there that debugging uncovered three pieces of buggy code, along with repairs to correct the bug. Combining these together, this produced the difference map:

action 1: **t** → **nil**

action 2: (*function* (cdr *variable*)) → t

action 3: nil → (*function* (cdr *variable*))

In TA, this difference map will be indexed by the inconsistencies between the old specification and the new one. In this example, there are three inconsistencies: the one relevant inconsistency of IF-ANY vs. FOR-ALL, and inconsistencies in names for the input variable and for the element of the IF-ANY and FOR-ALL instances.

Now, suppose TA is given the specification for ALL-NUMBERS?, which tests whether its single input l contains only numbers. This is specified as: (for-all x l (numeric x)). The corresponding program, also supplied to TA, is:

```
(cond ((null l) t)
      ((numberp (car l)) (all-numbers? (cdr l)))
      (t nil))
```

Suppose further that TA is given the specification for HAS-NUMBER?, which tests whether its single input l contains any numbers. This is specified as: (if-any x l (numeric x)). TA retrieves the plan just given for ALL-NUMBERS?.[11] TA cannot complete this plan successfully; however, the inconsistencies between HAS-NUMBER? and ALL-NUMBERS? match the inconsistencies between HAS-ATOM? and LAT? for the purposes of retrieval, so TA is able to retrieve the difference map created when writing HAS-ATOM?. This is applied and a correct program is generated.

4.3.5 Reorganization: Creating Schemas and Chunks

We continue with an example from TA to illustrate the creation of chunks, schemas, and their descriptions.

Suppose that TA is given the specification for FIRST-ATOM, which is to return the first atom in its single input *l*, and which is specified as (first-found x l (atomic x)). Along with this specification, TA is given the program:

```
(cond ((null l) nil)
      ((atom (car l)) (car l))
      (t (first-atom (cdr l))))
```

The associated plan is:

```
1 (->NULL QUESTION1 (COLLECTION (SPEC FUNCTION)))
```

[11] Actually, the plan for HAS-ATOM? matches just as well. TA breaks such ties in favor of cases more recently encountered.

```
2 (->NIL ACTION1)

3 (->CAR (ELEMENT (SPEC FUNCTION)) (COLLECTION (SPEC FUNCTION)))

4 (->ATOM QUESTION2 (ELEMENT (SPEC FUNCTION)) (TEST (SPEC FUNCTION)))

5 (->CAR ACTION2 (FIRST-ARGUMENT FUNCTION))

6 (->T QUESTION3)

7 (->CDR SYMBOL1 (FIRST-ARGUMENT FUNCTION))

8 (->RECURSE1 ACTION3 FUNCTION SYMBOL1)
```

After this, TA is asked to write FIRST-NON-ATOM, which is to return the first *non*-atom in its single input l, and which is specified to TA as (**first-found x l** (¬ (**atomic x**))). The plan for FIRST-ATOM is retrieved. In completion, one relevant inconsistency is discovered: FIRST-ATOM contains an instance of the **atomic** concept as the **test** of its **spec**, while FIRST-NON-ATOM contains an instance of ¬. There is only one step in the plan for FIRST-ATOM that is affected by this inconsistency, and that is step 4. TA's planner attempts to find a plan to replace the existing step 4; the best it can do is ->NOT. The state for ->NOT calls for an occurrence c of the concept ¬, a symbol, and lisp code that is the argument of c. The state for step 4 has two of these items, the symbol QUESTION2 and the instance of ¬ that is the **test** of the **spec** of FIRST-NON-ATOM, but the third element of step 4's state, the **element** of the **spec** of FIRST-NON-ATOM, is not the **argument** of the instance of ¬. Further planning is thus required, and a new plan is searched for whose state matches a newly created state consisting of those state elements not yet accounted for, namely the **argument** of the instance of ¬ (which is an instance of **atomic**) and the **element** of the **spec** of FIRST-NON-ATOM. ->ATOM, whose state calls for an occurrence c of the concept **atomic**, a symbol, and lisp code that is the argument of c, is found. This plan matches the new state, since the instance of **atomic** is a **symbol** (this because it is an **argument**, and **argument** $\stackrel{*}{\Rightarrow}$ **symbol**), and the **element** of the **spec** is both the **argument** of the instance of **atomic** and an instance of **lisp-code** (this last as a result of applying step 3 of the plan). Since these states match, no further planning is required, and the following two steps are added:

```
3.1 (->ATOM (ARGUMENT (TEST (SPEC FUNCTION)))
    (ELEMENT (SPEC FUNCTION)))

4 (->NOT QUESTION2 (ARGUMENT (TEST (SPEC FUNCTION)))
    (TEST (SPEC FUNCTION)))
```

Since the old plan step has been replaced with a sequence of steps, a chunk is created from these steps. The chunk is described by the four concepts used in deriving it, and those concepts are themselves given in terms of their roles in the plans that were used. Thus, the instance of ¬ is given as just that; QUESTION2 is given as a symbol, the instance of atomic is given as an argument, and the element of the spec is given as lisp-code. This description is used to index the chunk. The chunk itself is generalized to refer only to concepts appearing in its description:

```
1 (->ATOM (ARGUMENT ¬) (ARGUMENT (ARGUMENT ¬)))

2 (->NOT SYMBOL (ARGUMENT ¬) ¬)
```

Since completion was successful, a new schema is created:

```
1 (->NULL QUESTION1 (COLLECTION (SPEC FUNCTION)))

2 (->NIL ACTION1)

3 (->CAR (ELEMENT (SPEC FUNCTION)) (COLLECTION (SPEC FUNCTION)))

4 (* QUESTION2 (ELEMENT (SPEC FUNCTION)) (TEST (SPEC FUNCTION)))

5 (->CAR ACTION2 (FIRST-ARGUMENT FUNCTION))

6 (->T QUESTION3)

7 (->CDR SYMBOL1 (FIRST-ARGUMENT FUNCTION))

8 (->RECURSE1 ACTION3 FUNCTION SYMBOL1)
```

This corresponds to the program schema:

```
(cond ((null l) nil)
      ((function→spec→test (car l)) (car l))
      (t (function (cdr l))))
```

The plan schema is indexed by a generalization of the specifications for FIRST-ATOM and FIRST-NON-ATOM: (first-found x l (test x)). The concept test appears because it is the most specific concept that covers both the instance of atomic in FIRST-ATOM and the instance of ¬ in FIRST-NON-ATOM.

4.3.6 A Final Example

If the cases used in case-based problem solving were only of use in solving problems of similar scope, the approach would be somewhat limited. In the RECODER model, previously stored cases can be used as building blocks in solving more complex problems. We illustrate this with a final example from TA.

Suppose that TA, having seen all the examples presented thus far, is asked to write a program FIRST-ALL-NON-ATOM, which returns the first element x in its single input l such that no element of x is an atom. This is specified to TA as (first-found x l (for-all y x (¬ (atomic y)))). TA searches for a plan schema that matches, and retrieves the schema created when generating FIRST-NON-ATOM. Step 4 of this schema needs to be instantiated by generating a sequence of steps for the test of the first-found term. This test is an instance of for-all, which causes the schema created when generating ALL-EMPTY? to be retrieved.[12] The completed plan follows. Step 4 in the schema for first-found has been replaced with a chunk to retrieve and apply a schema; the role of spec in the chunk is played by the instance of for-all.

```
1 (->NULL QUESTION1 (COLLECTION (SPEC FUNCTION)))

2 (->NIL ACTION1)

3 (->CAR (ELEMENT (SPEC FUNCTION)) (COLLECTION (SPEC FUNCTION)))

    4.1 (GENERATE-SYMBOL FUNCTION91)
    4.2 (RETRIEVE-SCHEMA SPEC FUNCTION91)
    4.3 (APPLY-SCHEMA FUNCTION91)
    4.4 (GENERATE-CALL1 FUNCTION91 (ELEMENT (SPEC FUNCTION)))

5 (->CAR ACTION2 (FIRST-ARGUMENT FUNCTION)))

6 (->T QUESTION3)

7 (->CDR SYMBOL1 (FIRST-ARGUMENT FUNCTION))

8 (->RECURSE1 ACTION3 FUNCTION SYMBOL1)
```

This plan produces the program FIRST-ALL-NON-ATOM:

[12] This is implemented by redundantly storing each schema as a chunk. The chunk form includes additional steps to create an auxiliary program (by instantiating the schema) and to include a call to the auxiliary program in the original program.

```
(cond ((null l) nil)
      ((function91 (car l)) (car l))
      (t (first-all-non-atom (cdr l))))
```

Application of the chunk in the above plan causes the schema created when generating ALL-EMPTY? to be retrieved and completed. During completion, step 4 of the schema must be instantiated; this causes the chunk created when generating FIRST-NON-ATOM to be retrieved, with its ¬ concept replaced with a concept for the test of the spec of the function. The completed plan is shown below.

```
1 (->NULL QUESTION1 (COLLECTION (SPEC FUNCTION)))

2 (->T ACTION1)

3 (->CAR (ELEMENT (SPEC FUNCTION)) (COLLECTION (SPEC FUNCTION)))

    4.1 (->ATOM (ARGUMENT (TEST (SPEC FUNCTION)))
          (ARGUMENT (ARGUMENT (TEST (SPEC FUNCTION)))))

    4.2 (->NOT QUESTION2 (ARGUMENT (TEST (SPEC FUNCTION)))
          (TEST (SPEC FUNCTION)))

5 (->CDR SYMBOL1 (FIRST-ARGUMENT FUNCTION))

6 (->RECURSE1 ACTION2 FUNCTION SYMBOL1)

7 (->T QUESTION3)

8 (->NIL ACTION3)
```

This plan produces the program FUNCTION91:

```
(cond ((null l) t)
      ((not (atom (car l))) (function91 (cdr l)))
      (t nil))
```

CHAPTER 5

MEMORY ORGANIZATION

5.1 Introduction

This chapter, taken with appendix B, comprise the critique component of our thesis. We examine the role of generalization as it relates to storing and retrieving cases in a case-based planning system. We shall use the RECODER model to show that certain types of generalization are unnecessary in certain situations involving indexing of cases. Specifically, *concept clustering* techniques [Fisher, 1987] and the hierarchies of concepts that can be generated using such techniques [Lebowitz, 1986, Kolodner, 1984] are not necessary when dealing with what we refer to as *index matching* and *index assimilation.*

We begin by discussing the *indexing problem* in case-based reasoning, and isolate the parts of that problem with which we concern ourselves. Next, we describe how RECODER addresses these issues, and compare it with other approaches. The general points brought up in this chapter are supported by experiments discussed in appendix B.

5.2 The Indexing Problem

Broadly stated, the indexing problem is the problem of arranging case memory so that good cases can be found efficiently. In practice, there are several interpretations of what the "indexing problem" is:

- *The feature selection problem*: Given an instance characterized by a set of features, determine a set of features to use in searching memory.
- *The index matching problem*: Given a set of features, find one or more relevant cases.
- *The storage index selection problem*: Given a case characterized by a set of features, determine a set of features to be used as storage indices.

- *The index assimilation problem*: Given a set of storage indices, incorporate them into the existing case indices.

Each of these problems has received some attention. The feature selection problem is addressed in HYPO [Ashley, 1988], where *factual predicates* are used to determine a set of *dimensions*, which are used as features for searching. MBRtalk [Stanfill and Waltz, 1986] addresses the index matching problem, presenting an efficient parallel algorithm for finding relevant cases. [Barletta and Mark, 1988] approach the storage index selection problem with a technique for selecting storage indices derived from explanation-based learning [Mitchell *et al.*, 1986, DeJong and Mooney, 1986]. In [Bradtke and Lehnert, 1988], an approach called *coarse coding* is used in deriving search features and storage indices. CYRUS [Kolodner, 1984] and UNIMEM [Lebowitz, 1986] each describe concept clustering techniques for addressing the index assimilation problem.

RECODER's contribution to the indexing problem is in the areas of index matching and index assimilation. We are interested in a memory organization that allows both efficient retrieval of relevant cases and easy assimilation of new indices.

5.2.1 Work in Matching and Assimilation of Indices

Much of the current work in indexing builds on the ideas on *dynamic memory* presented in [Schank, 1982]. Schank's main claim is that memory is organized in *memory organization packets* (MOPs). MOPs are arranged in hierarchies along two dimensions: *abstraction* hierarchies define class-subclass relations, and *packaging* hierarchies define part-whole relations.[1]

[Schank, 1982] gives two examples of dynamic memory implementations, CYRUS and IPP. CYRUS is described more fully in [Kolodner, 1984], while the memory organization underlying IPP (known as UNIMEM) is described in [Lebowitz, 1986]. In each of these implementations, the abstraction hierarchy is implemented as an extended form of discrimination net [Feigenbaum, 1963, Charniak *et al.*, 1980], where:

1. Each node contains a set of features, which together define an abstract concept (a MOP).

2. Branches from nodes correspond to child concepts. Children are assumed to inherit all features from their parents.[2]

[1] Additionally, there is often an implicit order between the parts being packaged — the kinds of things that are generally packaged are events or abstractions of events.

[2] The requirement that all features be inherited is sometimes relaxed, in order to deal with noisy input. In UNIMEM, for example, there is a system parameter that allows one to specify the number of contradictions between parents and children that will be tolerated.

3. Leaves correspond to instances that have been encountered previously.

The index matching problem is addressed by comparing features of a current situation to those of nodes in the abstraction hierarchy. A search starts with the most general concept available. Features of a new instance are matched with features of concepts in the hierarchy (storage indices) until disagreements are found. The new instance is taken to be an instance of the parents of the disagreeing concepts (thus, searches can end at any node in the tree, not just at the leaves). This type of memory can be used in two basic ways:

1. To classify new instances.

2. To retrieve old instances.

For instance, in IPP, a concept was created having to do with kidnappings of Italian businessmen. This allowed new instances of, e.g., kidnappings of Italians to be classified as members of the same concept. This, in turn, allowed certain predictions to be made (e.g., that an Italian who is kidnapped will be a businessman).

On the other hand, CYRUS was used primarily as a question-answering system. CYRUS might be given a query such as: "Has Gromyko ever talked to Vance?" It would search for a concept in which this in fact occurred, and answer by listing the instances associated with the concept.

Another approach to the index matching problem makes use of *spreading activation* [Quillian, 1968, Hendler, 1988] or *relaxation* [Rumelhart *et al.*, 1986, Waltz and Pollack, 1985] algorithms. For instance, [Kolodner, 1988] describes a memory organization derived from that used in Direct Memory access Parsing [Riesbeck and Martin, 1985]. [Thagard *et al.*, 1988] describe an approach in which search is accomplished by building a constraint network and running a relaxation algorithm on it, much as is done in [Lehnert, 1987a] and [Lehnert, 1987b]. This line of work, however, has yet to deal with the index assimilation problem. In [Kolodner, 1988], a highly structured memory is assumed, with nothing said about how new cases are added to memory. In [Thagard *et al.*, 1988], the initial organization of memory is not an issue, but the technique requires constructing a constraint network, which has a time complexity quadratic in the number of cases.

In CYRUS and UNIMEM, the index assimilation problem is viewed as the problem of inserting new instances into the abstraction hierarchy. New instances are added by searching for concepts that classify them, and making them children of the concepts. If these concepts already index instances that are significantly similar to the new ones, generalizations are created and placed above the similar instances and below the parents.

These implementations are interesting first approximations, but leave some questions unanswered. First, it is often unclear what should be meant by "significantly similar." In [Lebowitz, 1986], instances are similar if they each share

some predetermined number of features, while not having very many features that contradict.[3] [Kolodner, 1984] uses a similar approach, and additionally uses domain knowledge when comparing features.

A more fundamental problem is the problem of controlling the generalizations made. This problem is illustrated well by Lebowitz's experiences with UNIMEM and its predecessor, IPP [Lebowitz, 1982, Lebowitz, 1988]. Lebowitz noted early in his work with IPP that that system tended to overgeneralize. His first attempt at a solution (discussed in [Lebowitz, 1982]) was to attach a confidence measure to each generalization. Confidence in a generalization was decreased whenever it encountered an instance that contradicted any of its features. When confidence in a generalization got too low, that generalization was deleted. This, however, created a new problem: useful generalizations tended to get deleted. Lebowitz solved this problem (in UNIMEM) by attaching a confidence measure to each *feature* of each instance. Whenever a feature's confidence got too low, it would be deleted from the generalization. This caused an entirely new problem, addressed in [Lebowitz, 1988]: generalizations that were initially distinct from one another could become the same by losing their distinguishing features. This could lead to redundancy not only in those generalizations, but in sub-hierarchies beneath the generalizations.

While it seems possible to control the size of the hierarchies created for specific sets of instances in specific domains, Lebowitz's experiences point out that the potential for creating explosively large hierarchies containing many incorrect, uninteresting or overlapping generalizations is very real.

In response to systems such as CYRUS and UNIMEM, Stanfill and Waltz propose a paradigm they call *memory-based reasoning* ([Stanfill and Waltz, 1986, Stanfill and Waltz, 1988]). According to Stanfill and Waltz, memory-based reasoning is an alternative to the type of approach described by [Lebowitz, 1986] and [Kolodner, 1984], which they call "similarity-based learning." Stanfill and Waltz see their work as achieving similar effects as those achieved by systems that learn rules based on abstracting similarities (as characterized by [Lebowitz, 1986]), but without needing the potentially expensive step of actually generating the rules. Instead, memory-based reasoning performs exhaustive parallel search of memory to extract the most similar cases, using a *similarity metric* to compute the similarity of each of the cases to the current situation.

We agree with Stanfill and Waltz that the implicit generalization provided by parallel partial matching is preferable to explicit generalizations of the sort described by [Lebowitz, 1986] and [Kolodner, 1984]. With UNIMEM and CYRUS, events are retrieved by matching features in a current situation with those in successively more specific generalizations; if the current situation is most similar to an event that happens not to be captured by an explicit generalization, that event will not be

[3] There is a closed world assumption — instances can only have a feature that contradicts if each instance is specified as having that feature

found. In MBRtalk it is as if the required generalization were created on demand, and the closest partial match (as given by the similarity metric) will always be found.

However, Stanfill and Waltz's examination of memory-based reasoning as an isolated phenomenon forces them to introduce some rather arbitrary and unnatural elements into their model. In particular, the similarity metric described in [Stanfill and Waltz, 1986] (and discussed in chapter 2) is overly complex and unintuitive. Our view is that the introduction of other forms of reasoning does away with the need for such unintuitive similarity metrics.

5.3 Matching and Assimilation of Indices in RECODER

5.3.1 Retrieval

There are four main assumptions behind the design of the retrieval component of RECODER:

1. There are many cases in memory.
2. There is no one "right" case to be used in solving a given problem.
3. A significant percentage of the cases in memory may be to some degree "relevant" to a given problem; it will in general be possible to impose a partial order of "relevance" on these cases.
4. A detailed analysis of a case is sufficiently resource-intensive that only a small number of cases should be subjected to it.

Thus, a retrieval mechanism that does detailed analysis on each case in memory, one at a time, is undesirable due to assumptions (1) and (4). Similarly, any mechanism that returns all relevant cases (and then, e.g., passes them on to an evaluation component which does one-at-a-time analysis) is also undesirable, due to assumptions (3) and (4). However, assumption (2) tells us that there may be any number of cases that could potentially benefit us; while assumption (3) tells us that there may be a way to order cases in such a way that we examine those that are most likely to benefit us first.

RECODER uses as its measure of relevance the degree of match between the state/goal information of the current problem and that of each of the cases in memory. Each case in memory is indexed by a number of features relating to state and goal information.

RECODER's retrieval mechanism as implemented in TA can be viewed broadly as consisting of two steps: *feature selection*, in which a set of search features is

derived from a problem situation, and *index matching*, in which the set of search features is compared to indices of prior cases.

5.3.1.1 Feature Selection

A problem situation is represented in TA as a composite concept. The feature selection phase transforms such a concept into a set of features, each consisting of a ***slot*** s_f and a ***value*** v_f. The algorithm responsible for this transformation, called *Transform*, transforms a concept c into a set of features f. *Transform* is also used to determine the set of features used as *indices* for cases. The algorithm is given below.

1. Let ***parts*** be all role/value pairs associated with c; i.e., ***parts*** $\equiv \{< x, y > \mid o \xrightarrow{x} y\}$. For `function` concepts, only the `spec` role is used to make up ***parts***.
2. For each $< p, v > \in$ ***parts***:
   (a) Find the set $S \equiv \{x \mid v \Rightarrow x \wedge x \Rightarrow p\}$. For each $x \in S$, return a feature f where $s_f = x$ and $v_f = v$.
   (b) If v can be decomposed (i.e., there exists concepts y and z such that $v \xrightarrow{y} z$), call *Transform* on v.

The feature selection mechanism can be illustrated with an example from TA. Suppose that TA's plan case base contains a plan for constructing the program `LAT?`, which tests whether its single input `list` is a list of atoms. `LAT?` is specified to TA as `(for-all x list (atomic x))`. This specification is represented internally as the concept shown in figure 5.1. This concept is decomposed into features, using *Transform*, as follows:

1. The `spec` role of `LAT?` is found to be played by `for-all117`. `for-all117` has one parent that is also a `spec`, namely `for-all`. Thus, `(for-all for-all117)` is returned as a feature. *Transform* is called on `for-all117`.
2. `for-all117` has three associated roles: `element`, `collection`, and `test`.
   (a) The role of `element` in `for-all117` is played by `x118`, whose only parent that is also an `element` is `element`. The feature `(element x118)` is returned.
   (b) The role of `collection` in `for-all117` is played by `list116`, whose only parent that is also a `collection` is `collection`. The feature `(collection list116)` is returned.

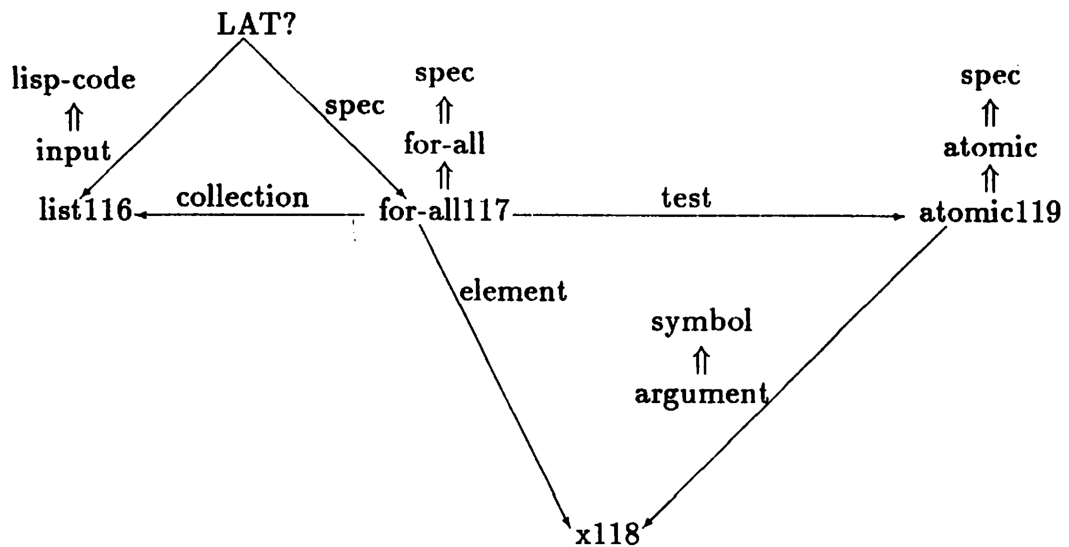

Figure 5.1: A network representation for LAT?.

(c) The role of **test** in **for-all117** is played by **atomic119**, whose only parent that is also a **test** is **test**. The feature **(test atomic119)** is returned.

3. Of the concepts **x118**, **list116**, and **atomic119**, only **atomic119** is decomposable, so *Transform* is called only on that concept. **atomic119** has only one role, **argument**, which is played by **x118**. The only parent of **x118** that is also an **argument** is **argument**. **(argument x118)** is returned as a feature. Since **x118** is not decomposable, the set of features is complete.

The complete set of features returned by *Transform*, and used to index the plan for **LAT?**, is:

(FOR-ALL FOR-ALL117)

(TEST ATOMIC119)

(COLLECTION LIST116)

(ELEMENT X118)

(ARGUMENT X118)

The complexity of *Transform* is nx^2, where x is number of parents of an "average" concept (assuming all concepts have roughly the same number of parents) and n is the total number of concepts associated with the representation of a

specification.[4] Step 1 can be performed in constant time for each concept, since each concept c contains a pointer to all pairs of concepts $< x, y >$ such that $c \xrightarrow{y} x$. The outer loop in step 2 will be iterated for each $p \in$ *parts*; it is at this point in the algorithm that all associated concepts of a specification will eventually be examined. Step 2.a is dominated by a nested loop. The outer loop is iterated $|S|$ times, once for each of the parents of the concept v; assuming that each concept has roughly x parents, this takes time x. The inner loop is iterated once for each parent of each member of S (i.e., once for each grandparent of v); it compares each grandparent to p and returns a new feature for each member of S with p as a grandparent. Again, assuming each concept has roughly x parents, the inner loop will take time x. Additionally, *Transform* will be called recursively on v if v can be decomposed (in step 2.b). There will be as many recursive calls as there are associated concepts in a specification.

5.3.1.2 Index Matching

Once a set of search features has been determined, index matching can commence. Index matching is a parallel process in which each case determines its degree of similarity to the current situation. A set of search features is broadcast (i.e., sent in parallel) to each of the potentially relevant cases. The cases then decide individually (in parallel) how well they match the problem situation, by counting the number of storage indices they have in common with the features sent them. Normalized versions of these estimates are then sent to a collection of *arbitration* nodes, which implicitly determines each case's position in a partial order of relevance to the current problem. Figure 5.2 shows a sample case base, with six cases connected to a tree of arbitration nodes.

There will, in general, be about the same number of arbitration nodes as cases. The arbitrators are arranged as an upside-down binary tree, with the leaves of the tree connected to at most two cases, and each internal node connected to at most two arbitrators. Estimates are sent from cases to the leaf arbitrators; these nodes identify the locally-best matches and send their results to their parent arbitrators. This process continues until the root arbitrator can suggest a globally-best match. Once the root arbitration node contains this globally-best match, that case can be

---

[4]The specification shown in figure 5.1 has five associated concepts: `LAT?`, `for-all117`, `list116`, `x118`, and `atomic119`. It would be more correct to say that $n =$ the total number of components of all concepts associated with a specification, but this number will in general be roughly the same as the total number of concepts associated with the specification, since concepts are associated with a specification by virtue of being components of other associated concepts. There may be more components than there are concepts, however, since it is possible for an encapsulated concept to be a component of more than one encapsulating concept. At present, this is only true of non-decomposable concepts; the algorithm would need to be modified to avoid redundant calls to *Transform* on decomposable concepts that have already been processed.

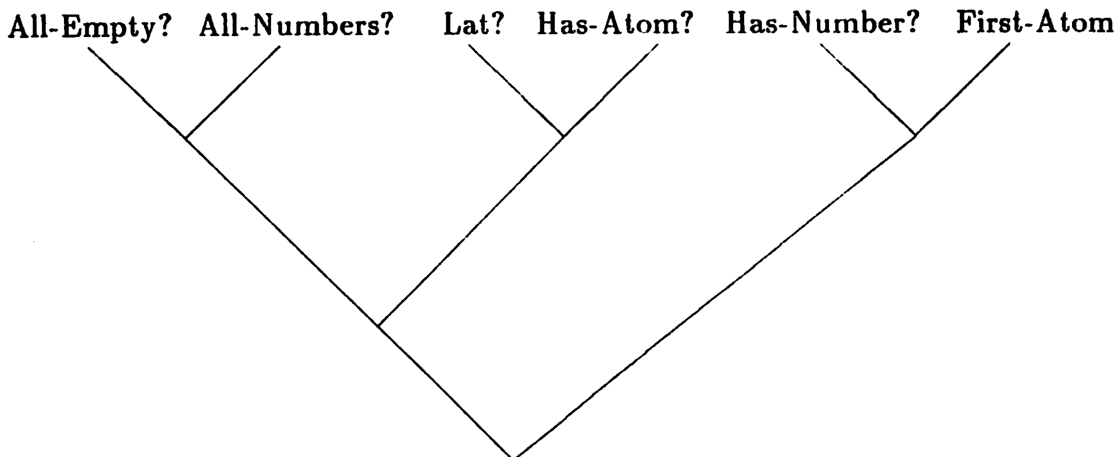

Figure 5.2: Case memory in TA

removed and the process can be repeated to find the next best match. If continued for all cases, this will result in a partial order of degree of match to the set of search features, and thus of relevance to the current problem.

The normalized estimate $N(s,c)$ for for computing the degree of match between search feature set s and case c is

$$N(s,c) \equiv \frac{\frac{M(s,c)}{F(s)} + \frac{M(s,c)}{F(c)}}{2}, \quad (5.1)$$

where $M(x,y)$ refers to the number of features in x for which there are matching slots with matching values in y, and $F(x)$ refers to the total number of features in x. As mentioned in chapter 2, the similarity metric computed by this formula is essentially the same as the *overlap metric* dismissed in [Stanfill and Waltz, 1986] as overly simplistic. It is indeed overly simplistic for memory-based reasoning, since the results of retrieval are subsequently used without modification; retrieval is thus burdened with finding a solution that is absolutely correct.[5] In RECODER, however, it is assumed that retrieval will find only a good first approximation to a solution; this will be refined with the aid of background knowledge.

The process of broadcasting a set of features to the cases in memory and then comparing those features to storage indices has a parallel complexity linear in the number of search features for the current problem. There are two operations involved for each search feature: broadcasting the index to cases, and determining if

[5] Memory-based reasoning assumes that initial problems will be decomposed into sub-problems. So while exact solutions to sub-problems must exist, there need be no exact solutions to the initial problems.

a feature matches a storage index for a given case. Broadcasting a feature to all cases can be done in constant time by, e.g., having an *input* node that is connected to all case nodes.[6] The process of each case determining if a given search index matches one of its storage indices also has constant parallel complexity. Each case can perform a table lookup on the index,[7] and these lookup operations can be done in parallel. Thus, all cases can determine whether they match a given feature in constant parallel time. Finding a degree of match for a set of features is a matter of counting the number of features that match, and thus has parallel complexity linear in the number of search features.

The process of finding the current most relevant case, once all cases have determined their degree of match, has parallel complexity logarithmic in the number of cases stored. The process of sending two estimates to an arbitrator and having the arbitrator determine which is the better match has constant complexity: there are two communications and one compare involved. Thus, the parallel complexity for each level in the tree of arbitrators is constant. The parallel complexity to determine the best match at the root of the arbitrator tree is thus equal to the depth of the tree, which is logarithmic in the number of cases.

5.3.2 Reorganization

Case memory is reorganized by adding new cases, indexed by relevant features. Cases are added and indexed as follows:

- Plans are added whenever old plans were retrieved and modified, whether by completion or through debugging. Plans are indexed directly by state and goal information.
- Schemas are added whenever they are created (which happens whenever completion yields a useful plan), and are indexed by common features of the retrieved case and the current problem.
- Difference maps are added whenever they are created, and are indexed by differences between the retrieved (buggy) case and the current problem.
- Chunks are added whenever they are created, and are indexed by the state information used to derive them.

---

[6]In practice, this would probably not be feasible. For example, given a hypercube architecture (such as the Connection Machine), it would take time logarithmic in the number of cases (assuming one case per processor) to broadcast a message.

[7]This assumes a known upper bound on the number of storage indices, and the ability to convert a search feature into a table lookup index in constant time.

Note that when considering any dynamic memory, it is important to consider the efficiency of adding new cases to memory. There are two considerations that are important in this regard:

1. The size of the memory (including indices and cases) should not grow uncontrollably.

2. The time it takes to insert a new case in memory should remain low.

The size of RECODER's case base will be strictly proportional to the number of cases plus the number of storage indices. Memory will consist of cases nodes and arbitrator nodes, and there will be about the same number of each. Each case node will contain the storage indices necessary for that node.

The time needed to insert a new case into memory will be logarithmic in the number of cases. The only time-consuming operation will be the addition of new arbitrator nodes, which can be done in time proportional to the depth of the arbitrator tree. This depth is logarithmic in the number of cases.

5.4 *Integrating Case Memory and Background Knowledge*

Most work in CBR assumes the existence of background knowledge apart from cases. In particular, in many CBR systems, reasoning is done about features of cases by incorporating those features into semantic hierarchies (see, for example, [Hammond, 1989, Sycara, 1987]). Even though these systems may exhibit learning in the form of generalizing features of cases, the background knowledge is assumed as given. Even systems that dynamically generate hierarchies of generalizations ("abstraction hierarchies," in the terminology of [Schank, 1982]) assume the pre-existence of similar but separate hierarchies containing abstractions of case features.[8]

It is, in fact, unlikely that humans acquire all their knowledge through the indiscriminate creation of similarity-based generalizations. Much of our knowledge is gained by "being told," whether through reading a book or through instruction by a teacher. We do not, for example, learn the names and usages of LISP functions by running random programs and generalizing the results. We are told the names and usages of these functions; generalization takes place when we attempt to put this knowledge to work writing programs to solve specific problems.

In RECODER, there is an intimate relation between cases and background knowledge. The features used to describe (and thus retrieve) cases are derived

[8] See, in particular, [Kolodner, 1984] and [Sycara, 1987]. In addition to general-to-specific relations, [Kolodner, 1984] assumes the pre-existence of "packaging" relations (terminology again due to [Schank, 1982]).

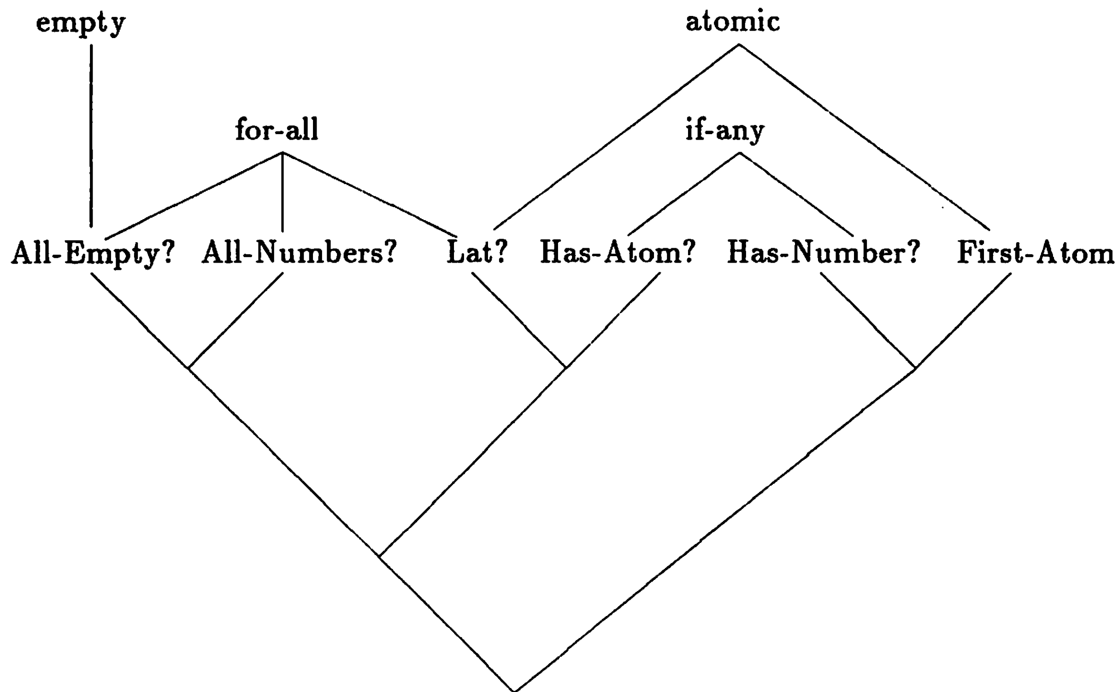

Figure 5.3: Cases linked to hierarchical relations

from concepts defined by pre-existing hierarchical knowledge. The relation can be made even more explicit by assuming direct links between background knowledge and cases, as in figure 5.3. Instead of converting specifications to features which are broadcast to cases, activation goes directly from specification concepts to cases. This has the flavor of Direct Memory Access Parsing [Riesbeck and Martin, 1985], where there is but a single representation for semantic relations and cases, and retrieval is accomplished through propagation of activation through a network.

Learning takes place quite naturally in such a scheme (see figure 5.4). New cases are assimilated into memory by adding links between background knowledge and cases. If cases are generalized into schemas, parts of their descriptions are generalized as well; this leads to links to schemas from more general background knowledge. RECODER's memory organization is thus a dynamically evolving one, but its dynamic nature is restricted to three types of learning:

1. Learning background knowledge by "being told."

2. Assimilating new cases by adding links from background knowledge.

3. Creating generalizations based on problem-solving experience.

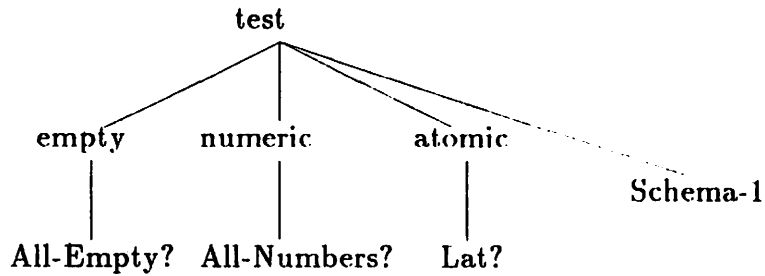

Figure 5.4: The effects of learning on prior knowledge

A notable omission from the above list is the type of learning most often associated with Dynamic Memory [Schank, 1982]: there is no learning through the creation of hierarchies of generalizations based solely on surface feature similarities. There is simply no reason to suppose that, if abstraction hierarchies are used by problem solvers, they are formed in this way. As appendix B demonstrates, a flat case organization along with a parallel partial-matching retrieval algorithm can achieve performance that rivals that of a generalization hierarchy used as a discrimination network. We have further argued here that even if we postulate the existence of abstraction hierarchies, we do not need to assume that these hierarchies are created through a process of indiscriminately comparing features and abstracting similarities. Instead, the majority of concepts and relations in such hierarchies will be learned by "being told," and the remaining generalized concepts will be added based on experience in solving problems.

CHAPTER 6

GENERALIZATIONS IN PROBLEM SOLVING

6.1 *Introduction*

In chapter 5 and appendix B, we argue that there is no need for explicit generalizations as indices to cases. Here, we propose that generalizations be used in the problem-solving process itself. We see generalizations as being useful in dealing with the *analogy problem*, that is, the problem of making a past case fit a current situation. The analogy problem can be broken down into three related problems:

1. *The evaluation problem.* This is the problem of evaluating a current situation in terms of a small number of prior cases, as in [Ashley and Rissland, 1986].

2. *The modification problem.* This involves methods for modifying past cases to fit current situations, as in [Alterman, 1988].

3. *The debugging problem.* The process of modification may overlook certain aspects of the current situation, causing failures to occur. The debugging problem is concerned with analyzing and minimizing these failures, as in [Hammond, 1987].

This chapter will demonstrate the use of generalizations with respect to the modification problem and the debugging problem. For the modification problem, we demonstrate how generalizations can improve the efficiency of RECODER's completion mechanism by comparing completion using generalizations with completion of the same plans using only past cases. For the debugging problem, we demonstrate how generalizations can be used as explanatory aid, by using schemas produced by RECODER to analyze buggy programs.

6.2 *Generalizations in Plan Modification*

In this section, we concentrate on using generalizations to deal with the modification problem. In particular, we examine how generalizations can be used to improve the performance of the case modification mechanism.

6.2.1 *Work in Analogical Problem Solving*

Before one can solve a problem by means of an analogy, one must establish establish what the analogy is. Gentner [Gentner, 1983, Gentner and Toupin, 1986] describes a *structure-mapping* theory of analogy, in which analogies between objects in a *base* domain and objects in a *target* domain are created by *systematically* mapping relations in the base domain to relations in the target domain. A mapping is systematic if the relations being mapped are themselves constrained by higher-order relations. These higher-order relations can themselves be mapped.

Analogies as described by Gentner are created using *structural* similarities of base and target concepts. Holyoak and Thagard [Holyoak and Thagard, 1989] argue that in addition to structural constraints, *semantic* constraints and *pragmatic* constraints are also important in creating analogies. Semantic constraints deal with similarity of meanings of relations in the base and target domains, while pragmatic constraints favor mappings that are deemed important by the agent making the analogy.

After analogies have been made, they can be put to use in solving problems. Carbonell [Carbonell, 1983, Carbonell, 1986] has identified two forms of reasoning from analogy: *analogical transformation* and *derivational analogy*. According to Carbonell, analogical transformation starts with solutions to past problems and transforms them to solutions to current problems, while derivational analogy uses the derivations of past solutions to construct new solutions.

[Shinn, 1988] describes a method of reasoning by analogy that he terms *abstractional analogy*. Given a prior case and a current problem, abstractional analogy will create an abstraction consistent with both. It will then apply the abstraction to the current situation to get a solution for the current problem. Abstractions are created either directly from previous solutions or from previous problem-solving methods.

[Alterman, 1988] describes a model of planning in which previously stored plans are adapted for use in a current problem situation. The approach, which Alterman refers to as *adaptive planning*, is quite knowledge-intensive, relying on detailed information about prior plans. Adapting a plan involves adapting each of the substeps of a plan in order. Each step is analyzed in detail for potential failures; if failures will occur, the step is adapted to the current situation (using prior knowledge).

[Rist, 1989] describes a model of planning in the domain of programming. In Rist's model, programmers proceed, if possible, in a *forward* direction, by using past programs or program schemas for current problems. If at some step no past knowledge is available, programmers proceed in a *backward* direction, dividing the goal of the program into subgoals, solving the subgoals, and recombining into a solution for the original goal.

6.2.2 *Solving Problems in RECODER*

The completion component of the RECODER model is responsible for detailed analysis of a case. RECODER's analysis consists of discovering inconsistencies

between the current problem and the case and modifying the case to account for the inconsistencies. Prior knowledge is used extensively during completion. Hierarchical relations are used in mapping the retrieved case description to the current problem (as in the structure-mapping model of [Gentner, 1983]) and using this mapping to determine inconsistencies between states and goals. Axiomatic plans are used in deriving new plan steps to modify the original case.

Completion begins by comparing the state and goal information of a retrieved plan to the current problem, and noting all inconsistencies between the two. Then, the plan is examined a step at a time. If state information for any step in the plan contains any of the aforementioned inconsistencies, the step is replaced with a sequence of steps derived from the relevant new state information, where possible. A retrieved plan that can be successfully completed is said to be ***useful***.

In completion, the derivation of new plan steps from axiomatic plans is closely related to planning of the sort done by STRIPS-style planners [Fikes and Nilsson, 1971]. However, the planning problem is simplified in that there is no planning from scratch — planning is always done as part of the completion process. This has two benefits. First, not all plan steps need to be replaced, only those that contain inconsistencies. Second, the structure of a new plan does not need to be derived, only some of the steps within that plan. This makes it possible to plan with incomplete knowledge.

In TA, the state and goal information for a planning situation is captured in the specification for a program, represented as a composite concept. TA's algorithm for finding inconsistencies between an old concept o and a new concept n is:

1. If $o \stackrel{*}{\Rightarrow} n$, then return an empty set of inconsistencies; otherwise:

2. Find the set of concepts $p_o \equiv \{x \mid o \Rightarrow x\}$ and the set of concepts $p_n \equiv \{x \mid n \Rightarrow x\}$.

3. If p_o is not identical to p_n, then return the set $\{(o, n)\}$; otherwise:

4. Find the set of concepts $c_o \equiv \{< x, y >\mid o \stackrel{y}{\rightarrow} x\}$ and the set of concepts $c_n \equiv \{< x, y >\mid n \stackrel{y}{\rightarrow} x\}$.

5. If $c_o = c_n = \emptyset$ and $n \stackrel{*}{\Rightarrow} o$, then return an empty set of inconsistencies; otherwise:

6. If $c_o = \emptyset$ or $c_n = \emptyset$ then return the set $\{(o, n)\}$; otherwise:

7. For each pair of elements $< x_o, y_o > \in c_o$, $< x_n, y_n > \in c_n$ such that $x_o = x_n$, return the union of inconsistencies between y_o and y_n.

The complexity of this algorithm is proportional to $m(x + y^2)$, where m is the number of associated concepts in a specification, x is the number of ancestors for

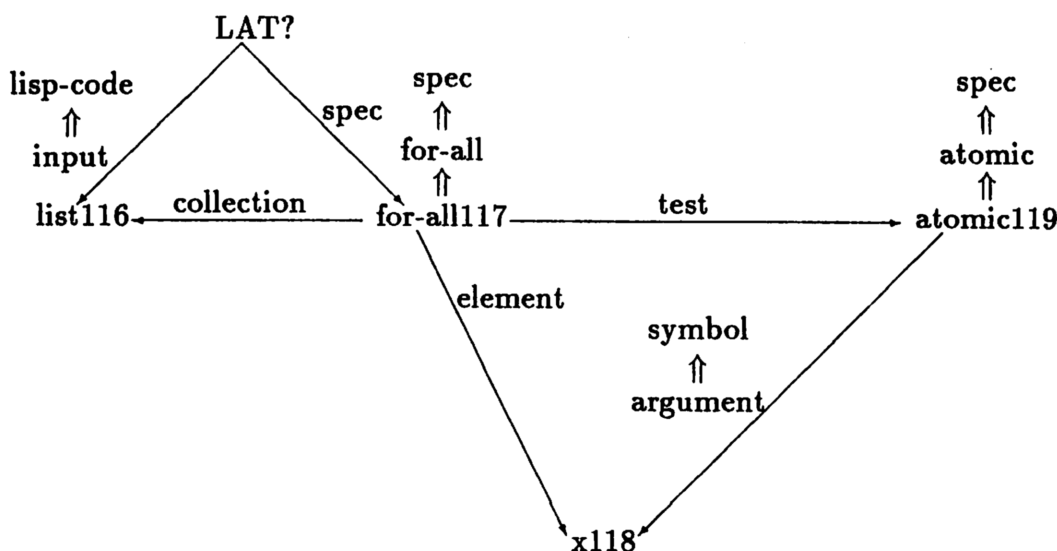

Figure 6.1: A network representation for a specification.

an "average" concept (assuming all concepts have roughly the same number of ancestors), and y is the number of components for an "average" concept (assuming all concepts have roughly the same number of components). Steps 1 through 7 above will be gone through once for each pair of concepts associated with the old and new specifications (in the specification shown in figure 6.1, there are five associated concepts: `LAT?`, `for-all117`, `list116`, `x118`, and `atomic119`). Steps 1 and 5 take time linear in the number of ancestors of a concept. Steps 2, 4, and 6 can be performed in constant time.[1] Step 3 can be done in time proportional to $|p_o| + |p_n|$; this is at worst proportional to x, since the number of parents of a concept will be less than or equal to the number of ancestors of that concept. Each of the recursive calls in step 7 can be performed in quadratic time using a nested loop. The outer loop steps through c_o, and the inner loop steps through c_n until a match is found. There will be as many recursive calls as there are associated concepts in a specification.

To illustrate the completion process, let us look at the TA program-writing system. Suppose TA is given a specification for a function `LAT?` that takes a list argument called `LIST`. `LAT?` is to be a predicate that returns `T` if `LIST` contains only atoms, `NIL` otherwise. This is specified to TA as `(for-all x list (atomic x))`. TA represents this specification internally as the semantic network shown in figure 6.1. Let us suppose further that TA is given the following program for `LAT?`:

```
(cond ((null list) t)
```

[1]Each concept c contains a pointer to all concepts x such that $c \Rightarrow x$, as well as a pointer to all pairs of concepts $< x, y >$ such that $c \xrightarrow{y} x$. It is assumed that testing for the empty set is a constant time operation.

```
((atom (car list)) (lat? (cdr list)))
(t nil))
```

TA creates the following plan to generate LAT?, and stores the plan in its case base.

```
1. (->NULL QUESTION121 (FIRST-ARGUMENT FUNCTION))
2. (->T ACTION122)
3. (->CAR (ELEMENT (SPEC FUNCTION)) (FIRST-ARGUMENT FUNCTION))
4. (->ATOM QUESTION124 (ELEMENT (SPEC FUNCTION)) (TEST (SPEC FUNCTION)))
5. (->CDR SYMBOL128 (FIRST-ARGUMENT FUNCTION))
6. (->RECURSE1 ACTION125 FUNCTION SYMBOL128)
7. (->T QUESTION130)
8. (->NIL ACTION131)
```

This plan can be paraphrased as:

1. Generate the COND question (NULL LIST), where LIST is the first argument of the function being written.

2. Generate the COND action T.

3. Generate the expression (CAR LIST); associate this expression with the element X of the spec of the function.

4. Generate, as the test of the spec of the function, the COND question (ATOM X); i.e. (from the previous association) (ATOM (CAR LIST)).

5. Associate the expression (CDR LIST) with the symbol SYMBOL128.

6. Generate as a COND action the recursive call of one argument (LAT? (CDR LIST)), from the function's name (LAT?) and SYMBOL128.

7. Generate the COND question T.

8. Generate the COND action NIL.

Now, suppose that TA is asked to write the program ALL-NON-EMPTY?, which again takes one argument LIST; but which tests whether or not all elements of LIST are non-empty (i.e., either non-nil atoms or lists containing at least one element). This is specified to TA as (for-all x l (¬ (atomic x))) and is represented internally as in figure 6.2.

TA begins by searching for a potentially relevant case; it finds the plan for LAT?.[2] Next, TA collects inconsistencies between specification for the old case

[2]Relevance is measured by equation 5.1, given in chapter 5.

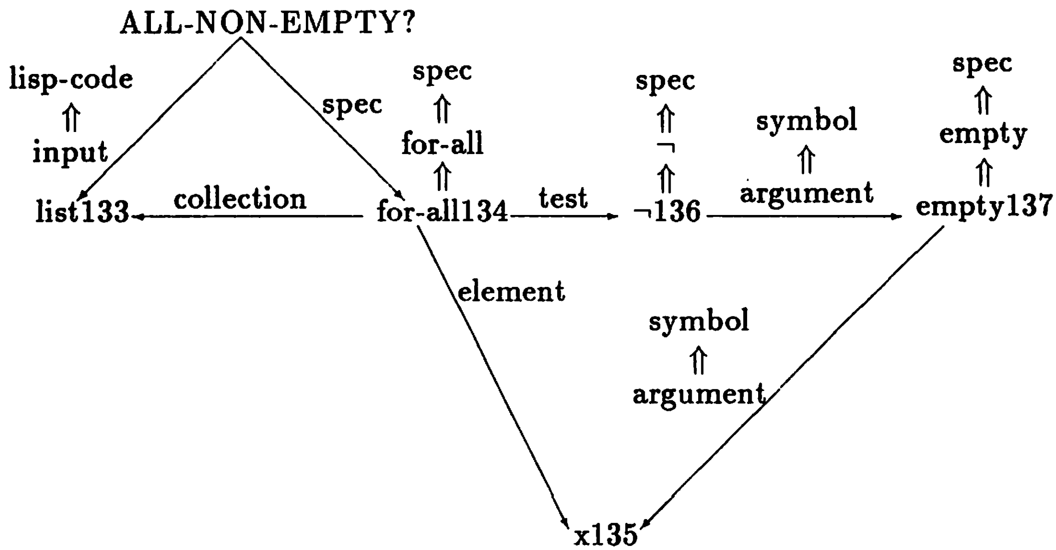

Figure 6.2: A second network representation for a specification.

and the new specification; the only important inconsistency is in the test of the specifications.[3] TA next examines each step of the old plan, looking for places in which the inconsistency occurs. The test of the specification occurs in only one step, namely step 4 of the original plan. Thus, step 4 is the only step to be rewritten.

TA rewrites this step by looking for an axiomatic plan whose state matches the analog in the new specification of the state of step 4; it finds the plan ->NOT. ->NOT requires a symbol, an occurrence of the specification concept ¬, and a piece of LISP code as an argument of ¬. The state of the step includes the symbol QUESTION124 and ¬136, the test of the spec of ALL-NON-EMPTY?, which is an instance of ¬. Unfortunately, the remaining element in the analog of the state of step 4, X135 (the element of the spec), is not the argument of the instance of ¬. This causes TA to attempt to find another potential plan, this one with a state matching the state containing X135 and the actual argument of ¬136, which is an instance of the specification concept EMPTY called EMPTY137. The plan ->NULL is found; this plan requires a symbol, an occurrence of EMPTY, and a piece of LISP code as the argument of EMPTY. EMPTY137 happens to serve as symbol and as an instance of EMPTY. X135 serves as LISP code, because of the association made in step 3 of the plan; it also happens to be the argument of EMPTY137. The result of running ->NULL is to associate the appropriate LISP code with the symbol in the plan's state, namely EMPTY137. Since EMPTY137 is also the argument of ¬136, the state conditions for ->NOT are also satisfied. Thus, step 4 in the original plan can be

[3]There are other inconsistencies, namely those arising from variables in different specifications being renamed apart. These are handled by merely substituting new variables for old in the final plan.

replaced by the two steps:

```
3.1 (->NULL (ARGUMENT (TEST (SPEC FUNCTION))) (ELEMENT (SPEC FUNCTION)))
4.  (->NOT (TEST (SPEC FUNCTION)) QUESTION124
          (ARGUMENT (TEST (SPEC FUNCTION))))
```

The modified plan produces the following program:

```
(COND
  ((NULL L133) T)
  ((NOT (NULL (CAR L133))) (ALL-NON-EMPTY? (CDR L133)))
  (T NIL))
```

At this point the completion process is over; however, TA goes on to create and store a plan schema encompassing both the old plan and the new one:

```
1. (->NULL QUESTION121 (FIRST-ARGUMENT FUNCTION))
2. (->T ACTION122)
3. (->CAR (ELEMENT (SPEC FUNCTION)) (FIRST-ARGUMENT FUNCTION))
4. (* QUESTION124 (ELEMENT (SPEC FUNCTION)) (TEST (SPEC FUNCTION)))
5. (->CDR SYMBOL128 (FIRST-ARGUMENT FUNCTION))
6. (->RECURSE1 ACTION125 FUNCTION SYMBOL128)
7. (->T QUESTION130)
8. (->NIL ACTION131)
```

The * in step 4 indicates that that step is general, and must be made concrete during completion in a way analogous to the way step 4 of the plan for LAT? was modified above. The new schema is indexed by the generalized specification **(for-all x 1 (*predicate* x))**, where *predicate* is the most specific concept that subsumes the conflicting specification concepts (i.e., the concept ¬136 and the corresponding concept in the older specification, ATOMIC119. The subsuming concept in this case is TEST).

Note that retrieval for plan schemas is slightly different from retrieval for plans. Plans can be retrieved whenever there is an intersection of problem features and case indices. Schemas can only be retrieved when all of their indices are present in a problem situation. But since schema indices contain more general concepts than plan indices (e.g., TEST is more general than ATOMIC or ¬), they allow schemas to be retrieved in a broader range of situations than plans that required exact matching of indices would.

6.2.3 *Using Generalizations to Focus Problem Solving*

Completion works by noticing differences between a past problem solving situation and a current one, and modifying steps in the past solution that reference those differences. This is in effect the same as constructing a generalization of the two situations, and using a similarly generalized solution as a basis for a new solution. Using an actual generalization means that a step can be skipped: RECODER does not need to discover a set of differences in the two situations and postulate a generalization that covers them, since a generalization is already available.

For example, suppose that TA were given a specification for `ALL-EMPTY?`, a program to test whether all elements of a single argument `LIST` were empty: `(for-all x list (empty x))`. Suppose further that the schema created in the last section were available to TA. This schema can be completed more efficiently than an actual case would be, since TA does not need to determine a set of inconsistencies and examine each step to find steps that contained one or more of those inconsistencies. Rather, TA assumes that all non-general steps will work as they appear; only general steps need to be examined. Thus, only step 4 of the schema in our example would be examined, and a process similar to the one described in the last section will yield the new step:

```
4. (->NULL (TEST (SPEC FUNCTION)) QUESTION124 (ELEMENT (SPEC FUNCTION)))
```

When step 4 of the plan schema is replaced with the newly created step 4, the following program results:

```
(COND
  ((NULL L53) T)
  ((NULL (CAR L53)) (ALL-EMPTY? (CDR L53)))
  (T NIL))
```

In order to determine the extent to which TA benefits from completion using schemas, 11 different programs were written both with and without the aid of schemas. The 11 specifications are shown in appendix A. The results of the experiment are summarized below.

Table 6.1 shows the number of steps in the plans for each of the programs being written, and the average time per step spent in completion. Each plan had the same number of steps when completed from a schema as it had when completed from a prior plan.

Table 6.2 compares the time spent in determining inconsistencies to the total time spent in completion for the programs completed from prior plans. As can be seen, the time spent in determining inconsistencies is quite small, never taking as much as 5% of the total time spent in completion.

Table 6.1: Total number of plan steps and average time spent in completion.

Steps	*Schemas*	*Plans*
17	5.70	9.18
18	10.11	11.72
18	9.33	14.17
21	10.81	16.43
22	13.82	21.73
23	16.39	25.61
31	12.42	19.39
25	15.32	24.76
24	18.75	29.04
37	14.49	21.89
34	18.36	27.35

Table 6.2: Time spent computing inconsistencies.

Inconsistencies	*Total time*	*Percentage*
6	156	3.85%
7	211	3.32%
12	255	4.70%
13	345	3.77%
16	478	3.35%
19	589	3.22%
13	601	2.16%
20	619	3.23%
23	697	3.30%
15	810	1.85%
20	930	2.15%

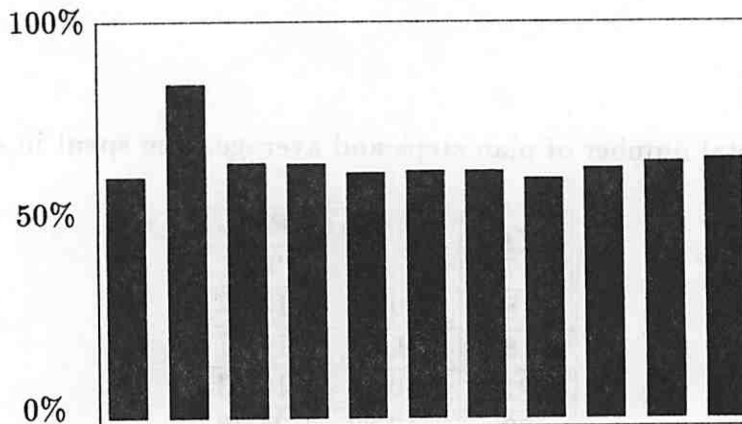

Figure 6.3: Speedup gained from completing schemas as opposed to plans.

Figure 6.3 shows graphically speedup gained in completing schemas as opposed to plans. The ratio of total time spent completing schemas to total time spent completing plans (excluding retrieval times in both instances) is shown for each of the 11 programs that were written. On average, completion from schemas is seen to take about 2/3 the time that completion from plans takes. Thus, while determining inconsistencies may have taken little time, checking each step in a prior plan for inconsistencies can add a significant amount of overhead to the total time spent in completion.

There are two parts to the completion process: examining steps (both to determine if modification is necessary and to add them to the new plan being created), and carrying out any necessary modifications. Table 6.3 and figure 6.4 show that, in general, the speedup gained from completing schemas comes from needing to do less examination. Table 6.3 compares the time spent modifying a schemas to the time time modifying prior plans. For a given program, the times spent actually doing modifications are fairly close, averaging out to a difference of about 2%, with a standard deviation of about 0.09 (from the ratios in the third column). Figure 6.4 shows graphically the ratio of times spent in examining steps for completion from schemas vs. plans; we can see that on average, examining steps in completion from schemas takes about 68% of the time as examining steps in completion from prior plans.

There are space considerations as well as time considerations. In RECODER, new plans are not saved if they result from successfully completing old plans or plan schemas. Thus redundant parts of plans are not duplicated in memory, as they would be if entire plans were always stored.

For example, TA has a training set of 35 program specifications. Using generalization on this training set results in a plan base containing 12 plans and a schema base containing 12 schemas, for a total of 24 cases. Without generalization, each

Table 6.3: Times spent in plan modification.

Schemas	*Plans*	*Ratio*
34	34	1.0
88	69	0.784
70	78	1.115
81	82	1.012
93	91	0.978
103	113	1.096
84	85	1.012
104	119	1.144
119	127	1.067
93	96	1.032
112	116	1.035

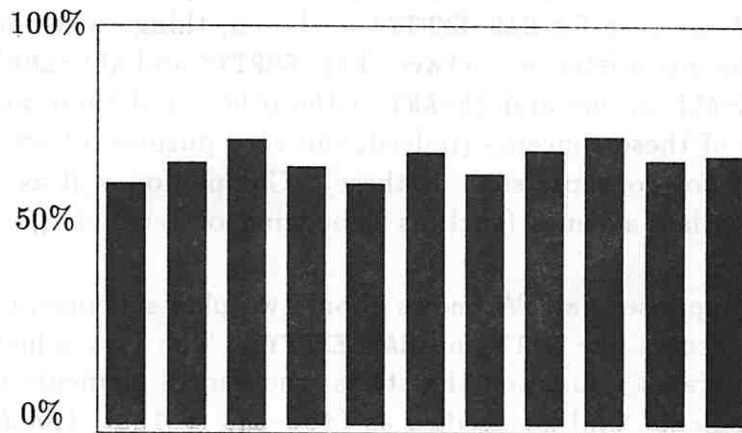

Figure 6.4: Speedup gained in examining schemas as opposed to plans.

new plan is saved, yielding a total of 35 stored plans.

It may be argued that, since one is already using completion to modify past cases, one need not generalize *or* store every case. One could, for example, store only those cases for which completion was not possible, and which could only be acquired through debugging or some other means of outside assistance. This would indeed cut down on space requirements, but it would introduce two undesirable side-effects. One side-effect we have already examined in depth: completing plans is more time intensive than is completing plan schemas.

The second side-effect concerns the relevance of the case retrieved to the problem at hand. When using plans, which are retrieved based on partial matches, there is no guarantee that a retrieved plan can be successfully completed. Schemas, however, are indexed by generalized case descriptions and can only be retrieved by exact matches; since generalizations are made only when completion has been successful, it is likely that, if a generalization is successfully retrieved, it will prove useful.

This can be illustrated with an example from TA. Suppose first that TA knows about two plans, each of which generates a function of one argument, assumed to be a list. The first function, `LAT?`, tests whether all elements in its argument are atoms, and has the specification `(for-all x list (atomic x))`. The second function, `HAS-EMPTY?`, tests whether any element in its argument is the empty list; it is specified as `(if-any x list (empty x))`. Now, suppose that TA is asked to write a function `ALL-EMPTY?`, which takes one argument (assumed to be a list) and tests whether all of its elements are the empty list. This is specified as `(for-all x list (empty x))`. By inspection, we can see that the specification for `ALL-EMPTY?` is as similar to `HAS-EMPTY?` as it is to `LAT?`, and in fact TA will make an arbitrary decision as to which specification will be examined first.[4] If it happens that the case for `LAT?` is retrieved, there will be no problem; completion will proceed as in the example in section 6.2.2. If the case for `HAS-EMPTY?` is chosen, things will not proceed quite so smoothly. The inconsistencies between `HAS-EMPTY?` and `ALL-EMPTY?` include the presence of `FOR-ALL` in one and `IF-ANY` in the other, and there are no axiomatic plans for either of these concepts (indeed, the very purpose of schema creation is to provide plans for concepts such as these). Completion will as a result not be successful, and other avenues (such as debugging or retrieving a different plan) must be tried.

Now, let us suppose that TA knows about two plan schemas, each generalized from completing cases like `LAT?` and `HAS-EMPTY?`. The first schema, which we'll call `ALL-X?`, generates a function that tests whether all elements in its argument satisfy some predicate, and is specified as `(for-all x list (`*predicate*` x))`. The second, called `HAS-X?`, tests whether any element in its argument satisfies some predicate, and is specified as `(if-any x list (`*predicate*` x))`. Let us again suppose

[4]The decision is not entirely arbitrary; the most recently stored case will be retrieved. But this choice has nothing to do with the content of the specifications.

Table 6.4: Comparing false hits in TA with and without schemas.

	Schemas	*Plans*
Total # of false hits	2	15
Instances containing false hits	1	5
Percentage of false hits	8.69 %	65.22 %
Percentage of instances with false hits	4.34 %	21.74 %

that TA is asked to write `ALL-EMPTY?`. Recall that retrieval of a plan schema requires each concept in its index to subsume a concept in the retrieval probe. This will result in retrieval returning `ALL-X?`, since each concept in that specification subsumes its corresponding concept in `ALL-EMPTY?`. `HAS-X?` will not be returned, since `IF-ANY` does not subsume `FOR-ALL`. `ALL-X?` can then be completed analogously to the example that began this section.

Table 6.4 gives some statistics for TA finding cases that it is unable to complete, for versions that did and did not store plan schemas. Each version was given 12 complete programs, and was tested on 27 additional programs. As can be seen, TA made close to an order of magnitude more mistakes when completing from past cases alone.

6.2.4 *Conclusions*

We have described the method used by RECODER to modify past cases to fit new situations, and have shown how generalized plans can aid in this process. Generalizations have seen to be useful in three ways: they decrease the time needed to modify cases, they conserve on space, and they focus search for relevant cases.

6.3 *Generalizations in Debugging*

This section will discuss RECODER's contributions to the debugging problem, particularly in the context of debugging *plan failures*. There are three broad issues relating to plan failures and debugging:

1. *Failure avoidance*. How can potential problem situations be avoided? This is a primary focus in [Hammond, 1986c, Hammond, 1987], where past failures are used repeating mistakes. [Luria, 1988] and [Alterman, 1988] describe knowledge-intensive ways of avoiding failures.

2. *Failure diagnosis.* What caused a failure to occur? Knowledge-intensive techniques for detecting failures are presented in [Hammond, 1987], [Spohrer *et al.*, 1985], and [Bonar, 1985]. [Shapiro, 1982] presents a set of interactive algorithms for detecting bugs in programs.

3. *Failure recovery.* How can a plan be modified to prevent future failures? This issue is addressed, in a case-based way, by [Hammond, 1986b]. [Shapiro, 1982] uses an inductive inference strategy for correcting bugs in programs.

RECODER contributes to all of the above areas, but its main contribution is in the area of failure diagnosis. We claim that important knowledge about diagnosing plan failures can be gained through experience in problem solving. In particular, RECODER can use its previously learned schemas to classify bugs in failed plans.

6.3.1 Work in Diagnosing Plan Failures

We see RECODER's ability to acquire diagnostic knowledge as an extension of work in diagnosing failed plans that relies on prior knowledge relating to diagnosis. One such example is CHEF [Hammond, 1987]. CHEF's plan diagnosis process, crucial to repairing and indexing plan failures, is:

1. Build an explanation of why a failure occurs. This is done by stepping through a trace of the plan simulation, asking a series of *explanation questions.*

2. Using the answers to the explanation questions, find an applicable *planning TOP.*[5] A planning TOP is a structure that corresponds to a particular type of abstract planning problem. Each planning TOP contains a set of repair strategies, as well as a set of features to be used as indices to failures.

Diagnosing plans in CHEF is an extremely knowledge-intensive process, requiring a fairly detailed simulation trace, a set of explanation questions, and a set of planning TOPS. The means for creating explanation traces, the explanation questions, and the planning TOPS must all be provided by a knowledge engineer.

Rather than presupposing extensive diagnostic knowledge, RECODER works under the assumption that such knowledge can be learned. Instead of creating an explanation trace and posing a set of explanation questions, RECODER steps through a trace of a plan simulation, obtaining advice interactively through questions of the form, "is this step correct?" As in [Shapiro, 1982], the bulk of diagnostic knowledge rests initially with the person interacting with RECODER, rather than with RECODER. But these interactions allow RECODER to form difference maps,

[5] *TOP* is short for Thematic Organization Point [Schank, 1982].

which can be used in future debugging situations much as planning TOPS are used in CHEF.[6]

Using RECODER to diagnose the planning problems of others builds on work done by [Bonar, 1985] and [Spohrer *et al.*, 1985] in diagnosing buggy novice programs. In [Bonar, 1985], there are identified two sources of knowledge used in programming: ***programming knowledge*** and ***step-by-step natural-language planning knowledge.*** Bugs result from the fragmentary nature of novices' programming knowledge. When gaps in this knowledge are encountered, they are bridged with buggy heuristics that Bonar refers to as ***bug generators.*** For example, one class of bug generators results from a novice's confounding structural similarities between programming knowledge and natural language planning knowledge with desired functional similarities. This could result in assuming that, e.g., control will exit from a WHILE loop ***as soon as*** the condition of the loop becomes false, rather than as soon as the condition is ***tested*** after it becomes false (which in languages like PASCAL happens only at the beginning of the loop). RECODER's view of debugging complements this by hypothesizing another knowledge source: cases of past programs. This results in a new class of bug generators, those caused by improperly modifying past cases.

[Spohrer *et al.*, 1985] characterizes bugs in programs in two ways: a ***problem-independent*** way, and a ***problem-dependent*** way. The problem independent categorization characterizes bugs along two dimensions:

- The ***component*** of a buggy plan that is in fact buggy. Seven types of components are identified. Example components are the input component, the initialization component, and the entire plan.

- The ***type of difference*** between the buggy plan and a correct version of the plan. There are four types of difference: missing component, malformed component, spurious component, and misplaced component.

The problem-dependent categorization adds to the above information obtained from a *GAP* (Goals And Plans) tree. A GAP tree can be thought of as a form of AND/OR tree, where AND nodes represent goals to be achieved and OR nodes represent possible plans for achieving those goals. Using such a tree has the effect of giving context to an error type. [Spohrer *et al.*, 1985] cite three ways in which this context can help:

- It allows a finer distinction of bugs.

- It gives the ability to pinpoint dependencies between bugs.

[6] Shapiro does not save the results of his interactions, though he does suggest the possibility of doing so.

- It makes it easier to identify *easy plans* (those in which bugs occur infrequently) and *hard plans* (those in which bugs often occur).

RECODER, too, allows for problem-dependent and problem-independent characterizations: plan schemas give a problem-dependent characterization, while difference maps give a problem-independent one. The primary contribution offered by RECODER is that RECODER offers a way of learning this diagnostic knowledge through experience in the domain.

6.3.2 Debugging in RECODER

Buggy plans can be characterized by their difference from ideal, correct plans, as follows:

1. A plan may have an incorrect step substituted for a correct one.
2. A plan may have a step deleted.
3. A plan may contain an unnecessary step.

In fact, the second and third situations are really subsumed by the first: a deletion is a substituted null step, and an addition is a substituted step where the original step was null.

Obviously, more than one of these situations could occur at once in a buggy plan. For example, consider a desired plan with steps x and y in the order x, y. Suppose a buggy plan contains these steps in the wrong order: y, x. This could be seen as the result of the step x being substituted for y, and the step y being substituted for x.[7]

In RECODER, buggy plans are discovered with the aid of *difference maps*. A difference map is a collection of pairs of *bugs* and *repairs*.

- A *bug* is a step description in a failed plan. In TA, these are discovered by stepping through traces of programs that failed, and determining (by asking for advice) which parts of the program were incorrect. This is similar to Shapiro's algorithm for diagnosing incorrect programs [Shapiro, 1982].
- A *repair* is a mapping from an incorrect plan step to a correct plan step. In TA, correct plan steps (i.e., program fragments) are obtained by asking for advice.

[7] Alternately, it could be seen as a result of the step x being deleted from its correct position before y and added to an incorrect position after y.

In general, the set of bugs in a difference map represents a characterization of a buggy plan, while the set of repairs represents what needs to be done to produce a correct plan.

Debugging in RECODER consists of either retrieving and applying a previously stored difference map or, if no relevant difference map can be found, tracing through a failed plan a step at a time, asking for advice when necessary. A debugging session that involves tracing a plan produces one difference map, with bugs and associated repairs corresponding to each failed step uncovered in the tracing process. New plans are generated by applying difference maps (whether retrieved or newly created) to buggy plans.

For example, consider the TA program-writing system, and the following four program specifications (specifications are given twice: once in English, and once in TA's specification language):

1. `LAT?`: "Is input *l* a list of atoms?"
   (for-all x l (atomic x))

2. `HAS-ATOM?`: "Does input *l* contain an atom?"
   (if-any x l (atomic x))

3. `ALL-NUMBERS?`: "Is input *l* a list of numbers?"
   (for-all x l (numeric x))

4. `HAS-NUMBER?`: "Does input *l* contain a number?"
   (if-any x l (numeric x))

Let us assume that the case base for difference maps is empty, and that the plan case base contains plans for the programs `LAT?` and `ALL-NUMBERS?`. The code for `LAT?` is as follows:

```
(cond ((null l) t)
      ((atom (car l)) (lat? (cdr l)))
      (t nil))
```

The code for `ALL-NUMBERS?` is:

```
(cond ((null l) t)
      ((numberp (car l)) (all-numbers? (cdr l)))
      (t nil))
```

Suppose that TA is asked to write `HAS-ATOM?`. TA examines its plan case base, and finds the plan for `LAT?`. It is not able to complete this plan, and so it enters debugging phase with the program for `LAT?` as its first approximation. No difference

maps are yet available, and so TA examines traces of program calls to find bugs and repairs. Eventually, bugs are found in each of the three actions of the `COND` clause, and the following difference map is created:[8]

action 1: `t` → `nil`

action 2: `(`*function* `(cdr` *variable*`))` → `t`

action 3: `nil` → `(`*function* `(cdr` *variable*`))`

Application of the difference map results in the following program:

```
(cond ((null l) nil)
      ((atom (car l)) t)
      (t (has-atom? (cdr l))))
```

The difference map and the new plan corresponding to the above program are stored in the appropriate case bases. The plan is indexed by the new specification, while the difference map is indexed by the differences between the old and new specifications:

old: `for-all`

new: `if-any`

Now, suppose TA is given the specification for `HAS-NUMBER?`. Both `ALL-NUMBERS?` and `HAS-ATOM?` match this specification equally well; let us suppose that TA chooses `ALL-NUMBERS?`. TA cannot complete this plan successfully; however, it is able to retrieve the difference map created when writing `HAS-ATOM?`. This is applied and the following correct program is generated:

```
(cond ((null l) nil)
      ((numberp (car l)) t)
      (t (has-number? (cdr l))))
```

The above scheme allows RECODER to deal with failure recovery and failure avoidance. Since RECODER remembers plans that it builds in response to failures, it can use these plans to avoid similar failures in the future. Since difference maps are also remembered, it can use these to recover from failures when they cannot be avoided. These solutions are similar to the ones offered in CHEF [Hammond, 1986b, Hammond, 1987].

---

[8]Italicized items denote variables.

In addition, RECODER can be used to diagnose the failed plans of others. When presented with a failed plan, RECODER can find a plan schema that matches the state/goal information for the plan. As in [Spohrer *et al.*, 1985], this can be used to give a ***problem-dependent*** characterization of the bug, i.e. a characterization in terms of the particular type of planning problem being attempted. This schema can then be compared to the buggy program to determine a ***problem-independent*** characterization, i.e. a characterization in terms of permutations of a correct plan. The next section will show how this can be done.

6.3.3 Using RECODER to Analyze Buggy Plans

This section is intended to demonstrate the plan analysis capabilities of RECODER, as realized in the TA programming system. To accomplish this, a number of fairly experienced LISP programmers were given a specification and asked to write a LISP program. The specification, resulting programs, and analyses are presented below.

6.3.3.1 The Experiment

The following specification we given to a number of graduate students who were also LISP programmers. These students for the most part write LISP code on a regular basis as part of their research. The specification follows.

> Write a function REPLACE with inputs OLD, NEW (both atoms) and LIST (a list). If LIST contains only atoms, REPLACE should replace each top-level occurrence of OLD with NEW in LIST; otherwise, REPLACE should replace all occurrences of OLD with NEW. REPLACE should call three auxiliary functions: one to do the test and one for each replacement method.
>
> Guidelines:
>
> 1. Use only the following LISP functions, special forms, and constants:
> ATOM, CAR, CDR, COND, CONS, EQ, NIL, NOT, NULL, T.
>
> 2. Do not run the code you have written.
>
> 3. Do not use any LISP references, including other people. Use only your own experience.
>
> 4. Don't spend more than 15 minutes on this.

Of the six programs received, only one worked with no modifications. Two others contained typographical errors: one contained a misspelled function name, the other missing parentheses. The remaining three contained semantic errors, which will be discussed below.

6.3.3.2 *The Programs*

A version of the above specification in TA's specification language would read as follows:

```
(if (for-all x l (atomic x))
    then (each x l (same o x) (replace n x (remainder l)))
    else ((deep each) x l (same o x) (replace n x (remainder l)))))
```

TA generated a set of four functions for the above specification, using a total of four previously created plan schemas. The top-level function is named `replace-dispatch`, and has three arguments: `o1164`, `n1165`, and `l1166`. Its code follows.

```
(COND
 ((FUNCTION90 L1166) (FUNCTION345 O1164 N1165 L1166))
 (T (FUNCTION448 O1164 N1165 L1166)))
```

The code for the function to test for atomicity is named `function90`, and has one argument, called `var91`. Its code follows.

```
(COND
 ((NULL VAR91) T)
 ((ATOM (CAR VAR91)) (FUNCTION90 (CDR VAR91)))
 (T NIL))
```

The function to replace only top-level occurrences is named `function345` and has three arguments `var346`, `var347`, and `var348`. Its code follows.

```
(COND
 ((NULL VAR348) NIL)
 ((EQ VAR346 (CAR VAR348))
  (CONS VAR347 (FUNCTION345 VAR346 VAR347 (CDR VAR348))))
 (T (CONS (CAR VAR348)
          (FUNCTION345 VAR346 VAR347 (CDR VAR348)))))
```

Finally, the remaining function is named `function448` and has arguments `VAR449`, `VAR450`, and `VAR451`. Its code follows.

```
(COND
 ((ATOM VAR451) VAR451)
 ((EQ VAR449 (CAR VAR451))
  (CONS VAR450 (FUNCTION448 VAR449 VAR450 (CDR VAR451))))
 (T (CONS (FUNCTION448 VAR449 VAR450 (CAR VAR451))
          (FUNCTION448 VAR449 VAR450 (CDR VAR451)))))
```

The overall form of a top-level function calling three other functions was fairly uniform throughout the six programs written by the subjects in the experiment, largely due to the restrictions placed on them by the specifications. Four subjects had virtually the same function as function90 above to test for atomicity. A fifth had a function that tested for the existence of a non-atom, called appropriately from the dispatch function. The sixth created an auxiliary function with a second argument containing a boolean value which kept track of whether or not all elements that had been examined at any given time were atoms.

The two functions designed to do the actual replacement fell into two broad classes. Four subjects coded these functions in a style quite similar to **function345** and **function448** above, though all four subjects interpreted the second function as dealing with a list of lists rather than a tree. This is perhaps understandable given that the specification states that the argument is to be traversed is a list; however, it is not stated whether or not dotted pairs are allowed within the list. The subjects all made the assumption that dotted pairs would not be present. This assumption can in fact be explicit in TA with the specification ((listified (deep each)) x l (same o x) (replace n x (remainder l))). This specification will generate a function here called function1529, with arguments VAR1530, VAR1531, and VAR1532:

```
(COND
  ((NULL VAR1532) NIL)
  ((EQ VAR1530 (CAR VAR1532))
   (CONS VAR1531 (FUNCTION1529 VAR1530 VAR1531 (CDR VAR1532))))
  ((ATOM (CAR VAR1532))
   (CONS (CAR VAR1532) (FUNCTION1529 VAR1530 VAR1531 (CDR VAR1532))))
  (T
   (CONS (FUNCTION1529 VAR1530 VAR1531 (CAR VAR1532))
         (FUNCTION1529 VAR1530 VAR1531 (CDR VAR1532)))))
```

Each of the subjects who took this approach wrote a function quite similar to function1529.

The other two subjects made an assumption of a different sort: they each wrote their replacement functions in a style that optimizes for tail recursion. For functions involving a single recursive call, this style involves arranging things so that the recursive call is the last thing done. This generally involves the use of an extra argument to accumulate intermediate results. Further, this style involves certain list manipulations that aren't necessary in the non-tail-recursive versions. There are two ways to accomplish these manipulations: one uses reverse and the other uses append:[9] In TA, the version using append has the specification ((tail-append

[9]Interestingly, neither reverse nor append were included in the list of functions allowed to be used. Both subjects seemed to justify their inclusion on the grounds that they were easily written

each) x l (same o x) (replace n x (remainder l))) and generates the function **subst-t1**:

```
(COND
  ((NULL L681) R682)
  ((EQ 0679 (CAR L681))
   (SUBST-T1 0679 N680 (CDR L681) (APPEND R682 (CONS N680 NIL))))
  (T (SUBST-T1 0679 N680 (CDR L681) (APPEND R682 (LIST (CAR L681))))))
```

The version using **reverse** has the specification **((tail-reverse each) x l (same o x) (replace n x (remainder l)))** and generates the function **subst-t2**:

```
(COND
  ((NULL L799) (REVERSE R800))
  ((EQ 0797 (CAR L799))
   (SUBST-T2 0797 N798 (CDR L799) (APPEND (CONS N798 NIL) R800)))
  (T (SUBST-T2 0797 N798 (CDR L799) (CONS (CAR L799) R800))))
```

One of the subjects writing in a tail-recursive style used **append**, while the other used reverse. Their functions are similar to the ones generated by TA.

Interestingly, each subject coding in a tail-recursive style for their function that replaced in a list of atoms carried that style over to their function that replaced in a list containing non-atoms. This is interesting for two reasons: one, the resulting functions aren't truly tail-recursive, and two, both of these functions contained errors. In addition, this approach makes the assumption made by the other subjects, i.e. that the list in which replacements are made contains no dotted pairs. The version of this approach using **append** can be specified in TA as **((tail-append (deep each)) x l (same o x) (replace n x (remainder l)))** with the function **subst*-t1**:

```
(COND
  ((NULL L927) R928)
  ((EQ 0925 (CAR L927))
   (SUBST*-T1 0925 N926 (CDR L927) (APPEND R928 (CONS N926 NIL))))
  ((ATOM (CAR L927))
   (SUBST*-T1 0925 N926 (CDR L927) (APPEND R928 (LIST (CAR L927)))))
  (T
   (APPEND R928 (CONS (SUBST*-T1 0925 N926 (CAR L927) NIL)
                      (SUBST*-T1 0925 N926 (CDR L927) NIL)))))
```

The version using **reverse** has the specification **((tail-reverse (deep each)) x l (same o x) (replace n x (remainder l)))** and generates the function **subst*-t2**:

---

using the allowed functions.

```
(COND
  ((NULL L1059) (REVERSE R1060))
  ((EQ O1057 (CAR L1059))
   (SUBST*-T2 O1057 N1058 (CDR L1059) (APPEND (CONS N1058 NIL) R1060)))
  ((ATOM (CAR L1059))
   (SUBST*-T2 O1057 N1058 (CDR L1059) (CONS (CAR L1059) R1060)))
  (T (SUBST*-T2 O1057 N1058 (CDR L1059)
                (CONS (SUBST*-T2 O1057 N1058 (CAR L1059) NIL) R1060))))
```

These are similar to the functions written by the subjects.

6.3.3.3 TA's Analysis

TA was used to analyze programming errors in the following way. Specifications were associated with buggy programs, which were transformed to corresponding plans. These specifications were then used to retrieve plan schemas. The plan schemas were then compared to the buggy plans, with differences noted.

There were three semantic errors made by the subjects, each of which was analyzed by TA. The first error was made by two different subjects, and each subject made the error in both replacement functions.[10] We shall examine the error as it appeared in a program for the specification `((tail-append each) x lst (same old x) (replace new x (remainder lst)))`. The buggy program, here called `replace-list-1`, was:[11]

```
(cond ((null lst) result)
      ((eq old (car lst))
         (replace-list-1 old new (cdr lst)
            (append result (cons (car lst) nil))))
      (t (replace-list-1 old new (cdr lst)
            (append result (list old)))))
```

TA found a plan schema for the generalized specification `((tail-append each) x lst` *test message*); the corresponding program schema follows.[12]

```
(cond ((null lst) result)
      ((function→spec→test old (car lst))
         (function old new (cdr lst)
```

[10]A third subject commented later that they almost made the same error, but noticed it in time.

[11]The original program, as with the originals for the other buggy programs presented here, was in a slightly different form. Syntactic modifications were made to make the program compatible with TA.

[12]Italicized items in specifications denote variables.

```
            (append result (function→spec→message (car lst) nil)))
      (t (function old new (cdr lst)
            (append result (list (car lst))))))
```

Comparing this schema with the program for `replace-list-1`, TA discovered a discrepancy: the last expression in `replace-list-1` is `(list old)`; in the schema, this expression is `(list (car lst))`.[13]

The second error was in a program for the specification `((tail-append (deep each)) x lst (same old x) (replace new x (remainder lst)))`. The buggy program is here called `replace-tree-1`:

```
(cond ((null lst) result)
      ((eq old (car lst))
         (replace-tree-1 old new (cdr lst)
            (append result (cons (car lst) nil))))
      ((atom (car lst))
         (replace-tree-1 old new (cdr lst)
            (append result (list (car lst)))))
      (t (append result
            (append (replace-tree-1 old new (car lst) nil)
               (replace-tree-1 old new (cdr lst) nil)))))
```

TA found the following schema for the specification `((tail-append (deep each)) x lst` *test message*`)`:

```
(cond ((null lst) result)
      ((function→spec→test old (car lst))
         (function old new (cdr lst)
            (append result (function→spec→message (car lst) nil))))
      ((atom (car lst))
         (function old new (cdr lst)
            (append result (list (car lst)))))
      (t (append result
            (cons (function old new (car lst) nil)
               (function old new (cdr lst) nil)))))
```

On comparison, `replace-tree-1` was found to have a call to `append` where it should have had a call to `cons`.

---

[13]TA discovers such discrepancies by finding syntactic differences in plans. Thus, it looked at the plans that generated each of the programs, and noticed that the step to generate a car expression was missing in the flawed plan.

The third error was in a program for the specification ((tail-reverse (deep each)) x lst (same old x) (replace new x (remainder lst))). The buggy program is here called replace-tree-2:

```
(cond ((null lst) (reverse result))
      ((eq (car lst) old)
         (replace-tree-2 old new (cdr lst)
            (append (cons (car lst) nil) result)))
      ((atom (car lst))
         (replace-tree-2 old new (cdr lst)
            (cons (car lst) result)))
      (t (cons (replace-tree-2 old new (cdr lst)
            (replace-tree-2 old new (car lst) nil)))))
```

TA found the following schema for the specification ((tail-reverse (deep each)) x lst *test message*):

```
(cond ((null lst) (reverse result))
      ((function→spec→test old (car lst))
         (function old new (cdr lst)
            (append (function→spec→message (car lst) nil) result)))
      ((atom (car lst))
         (function old new (cdr lst)
            (cons (car lst) result)))
      (t (function old new (cdr lst)
         (cons (function old new (car lst) nil) result))))
```

On comparison, replace-tree-2 was found to have the order of the final call to cons and the immediately following recursive call reversed.

6.3.4 Conclusions

Though the sample of programs analyzed by RECODER is small, some interesting results were obtained. First, the method of analyzing failed plans by using generalizations created from cases shows promise, and indeed that it was useful in analyzing the plans above. Given a case-based model of programming, one would assume that programs that proved difficult to write would be those for which cases were not available to the programmer. Indeed, these were the very programs that proved most difficult for the subjects. The subjects using a tail-recursive style of programming were taught this style for use with programs that operate on the top-level elements of a list, but not for programs that operate on a list as if it were a tree; there is no advantage of using this style of programming for general tree structures. Nonetheless, the subjects using this approach attempted to modify

their list-oriented schema for tail recursion to deal with trees, and neither subject making this attempt succeeded.

The very fact that these subjects chose to program in this style at all is evidence that prior experience plays an important role in problem solving. Both subjects had recently completed a LISP programming class in which the tail-recursive style of programming was emphasized; this emphasis carried over to their participation in the experiment. Indeed, one of the subjects remarked that they did not make a conscious decision to program in that style, and did not even think of the efficiency issue; it was simply the first approach that occurred to them.

Additionally, this experiment demonstrates the prominent position of debugging in the problem-solving cycle. Of the six programs received, only one contained no errors. Several subjects commented on the difficulty in writing programs without being able to run them. As mentioned earlier, the subjects in this experiment were not novices; they had all had moderate to extensive LISP programming experience. This suggests that even with experience, making and correcting mistakes is an important part of solving problems.

The experiment also shows a direction for further work regarding debugging in RECODER. Currently, the debugging component operates under the assumption that if an error has been made, it is the result of using a plan that could not be completed properly, and thus was used in its original form. Thus, whenever TA encounters a situation during completion that it cannot handle, it abandons the completion process and debugs the originally retrieved program.

The experiment shows that this is not a realistic assumption to make. All three of the errors discussed above were caused not by using inappropriate plans, but by completing plans incorrectly. In the first program, the programmer apparently assumed that because the expressions `old` and `(car lst)` were interchangeable in one part of the program, they were therefore interchangeable throughout the program. In the second and third programs, the programmers apparently tried to force schemas for tail recursion in lists (which never need to recurse on the **car** of a list) into situations involving trees (that is, where it is necessary to recurse on the **car** of a list). These programmers most probably had separate schemas for dealing with tree recursion; attempting to, in effect, create a new schema by cobbling together two existing schemas produced the buggy programs shown.

Chapter 7

Analytic Concept Creation

7.1 Introduction

This chapter will describe *analytic concept creation*, the technique used by RECODER to create and index plan schemas. The technique can be seen as an extension of *concept clustering* [Fisher, 1987] that makes use of the *analytic paradigm* [Carbonell, 1989]. It is uniquely suited to the task of generalizing the results of case-based planning, since

1. Completing old plans to fit new situations is an analytic task (at least in the RECODER model), and
2. Completion transforms *old* plans into *new* ones; clustering these old and new plans is a natural way to form generalizations.

We shall proceed by first describing the learning problem in the context of case-based reasoning, and discussing related learning work. We shall then describe RECODER's generalization technique, and show how this technique compares with more traditional clustering techniques.

7.2 The Learning Problem

The learning problem for case-based reasoning can be broken down into three subproblems:

- How are cases assimilated into case memory?
- How and when are cases generalized?
- How and when are case indices generalized?

The first subproblem is the index assimilation problem, discussed in chapter 5. As to when cases and indices are generalized, we suggest generalizations that arise from successes and failures of the problem solver. But rather than simply classifying instances as positive and negative exemplars as in Lex [Mitchell *et al.*, 1983] or decision tree approaches [Quinlan, 1983], classes are created that capture similarities between the problem that was solved and the past case that was used to contribute to the solution. Thus, RECODER's method of learning can be seen as combining methods in which instances are classified into predefined classes with conceptual clustering methods [Fisher, 1987, Lebowitz, 1986], which create new classes.

7.3 Related Learning Work

RECODER's primary technique for generalizing indices has its roots in concept clustering techniques such as those described in [Kolodner, 1984, Lebowitz, 1986, Fisher, 1987]. The main extension to those works involves the approach taken in addressing the *clustering* problem. According to [Fisher, 1987],

> The *clustering* problem involves determining useful subsets of an object set. This consists of identifying a set of object classes, each defined as an extensional set of objects.

In CYRUS [Kolodner, 1984] and UNIMEM [Lebowitz, 1986], heuristics are used in determining when instances are similar enough to merit forming new classes. In COBWEB [Fisher, 1987], a metric called *category utility* is used both to classify objects and to decide when new classes should be created. RECODER uses the results of its problem solving component to aid in its classifications, as will be described below.

Using the results of problem solving to determine when generalizations should be made is related to ideas in explanation-based learning [Mitchell *et al.*, 1986, DeJong and Mooney, 1986]. In learning from explanations, the important notion is that successes in solving problems (or, in [Mitchell *et al.*, 1986], successes in proving theorems) can be used to aid in learning. For Mitchell et. al, the explanation-based learning (EBL) problem assumes the following as given:

- A goal concept (expressed in a non-operational manner).
- A training example.
- A domain theory (a set of rules and facts used in explaining the training example in terms of the goal concept).
- An operationality criterion, specifying the form in which the learned concept is to be expressed.

The problem is to find a generalization of the example that is a sufficient concept definition for the goal concept and that satisfies the operationality criterion. The ability to generalize in this manner follows directly from the ability to explain the training example in terms of the goal concept. This explanation is in fact a proof in the domain theory that the example follows the goal concept definition.

[DeJong and Mooney, 1986] express the EBL problem in terms of problem solving rather than theorem proving. For them, the following is assumed as given:

- A domain theory (including *object specifications*, *relations between objects*, and *operators* and *schemata*).
- A goal.
- An initial world state.
- A sequence of operators that achieves the goal from the initial state (if explanations are to be obtained through observation).

The problem is to determine a new schema that achieves the goal in a general way.

7.4 Generalization in RECODER

7.4.1 Types of Generalization in RECODER

As mentioned above, two kinds of generalization must be considered in case-based reasoners that learn: generalization of the cases themselves, and generalization of indices to cases. In RECODER, cases are generalized in three ways:

- When completion yields a useful plan, a generalization is made of the new plan and the plan used in the completion process. This is done by abstracting out sequences of steps that differ between the two plans, leaving enough information for the steps to be completed in the future. The result is a plan *schema*.
- When completion generates new plan steps, they are compiled into *chunks* [Laird *et al.*, 1986] with some constants generalized into variables. Chunking has its roots in work done with STRIPS [Fikes *et al.*, 1972], and is closely related to the idea of *macro-operators* [Korf, 1985].
- When debugging occurs, a *difference map* is constructed. This is a description of a class of plans that will fail in a particular way, and as such constitutes a generalization over those plans.

The idea of using the success or failure of a solution to dictate positive and negative exemplars for generalization of cases has received some attention [Carbonell, 1983], but RECODER extends this with a technique for creating concept descriptions that can be used to index generalizations. In RECODER, each case is described by a set of features, some subset of which are used to index the case. Whenever two plans are merged to form a plan schema, their features are also merged to form a generalized description to index the new plan schema; this creates a new class in the space of case indices.The situation is similar for difference maps, though there the set of features used as indices consists of differences between features of the two plans in question. Indices for chunks are determined using explanation-based methods [Mitchell *et al.*, 1986, DeJong and Mooney, 1986]. In RECODER, as in [DeJong and Mooney, 1986], explanations are viewed as paths in a search space, as opposed to the more restricted view given in [Mitchell *et al.*, 1986] requiring explanations to be proofs in logic. The search paths in RECODER do not represent complete solutions to problems, but only the partial solutions resulting from completion. These search paths are treated as chunks, and are indexed by the results of performing explanation-based generalization on them (this technique is similar to that used in [Barletta and Mark, 1988]).

While each of the above three methods of generalization have been implemented in TA, the focus of this work has been on the creation of schemas and schema descriptions. The following section will describe how schemas and their descriptions are created in TA.

7.4.2 Creating Schemas and Schema Descriptions in TA

A schema is created by abstracting out the differences between a retrieved plan and a plan that has been successfully completed; a schema description is similarly created from old and new plan descriptions. Schema creation itself is a fairly straightforward process, consisting of replacing modified steps of the retrieved plan with variabalized steps. This process is linear in the number of steps. The creation of a schema description is more complex, involving abstracting out inconsistencies in the old and new plan descriptions. The algorithm for creating a schema description follows.

1. Generate a new concept g.
2. Find the set of concepts $p_o \equiv \{x \mid o \Rightarrow x\}$ and the set of concepts $p_n \equiv \{x \mid n \Rightarrow x\}$.
3. Create the set of concepts $p \equiv p_o \cap p_n$.
4. For each $x \in p$, let $g \Rightarrow x$.

5. Find the set of concepts $c_o \equiv \{< x, y > \mid o \xrightarrow{x} y\}$ and the set of concepts $c_n \equiv \{< x, y > \mid n \xrightarrow{x} y\}$.

6. Create the set of concepts $c \equiv \{< x, y, z > \mid < x, y > \in c_o \wedge < x, z > \in c_n\}$.

7. For each $< x, y, z > \in c$, create $r \equiv$ generalize(y, z) and let $g \xrightarrow{x} r$.

The complexity of this algorithm is proportional to $m(x + y^2)$, where m is the number of associated concepts in a specification, x is the number of parents for an "average" concept (assuming all concepts have roughly the same number of parents), and y is the number of components for an "average" concept (assuming all concepts have roughly the same number of components). Steps 1 through 7 above will be gone through once for each pair of concepts associated with the old and new specifications (in the specification shown in figure 7.1, there are five associated concepts: LAT?, for-all117, list116, x118, and atomic119). Steps 1, 2, and 5 take constant time for each associated pair of concepts. Step 3 can be done in time proportional to $|p_o| + |p_n|$,[1] which is proportional to x, and step 4 can be done in time x. Step 6 can be done in time proportional to $(|c_o| + |c_n|)^2$, which is proportional to y^2, using a nested loop. The outer loop steps through c_o, and the inner loop steps through c_n until a match is found. There will be as many recursive calls (step 7) as there are associated concepts in a specification.

The following example illustrates schema creation in TA. Suppose TA is given a specification for a function LAT? that takes a list argument called LIST. LAT? is to be a predicate that returns T if LIST contains only atoms, NIL otherwise. This is specified to TA as (for-all x list (atomic x)). TA represents this specification internally as the semantic network shown in figure 7.1; this description is used to index the plan to write LAT?, which follows.

```
1 (->NULL QUESTION1 (COLLECTION (SPEC FUNCTION)))

2 (->T ACTION1)

3 (->CAR (ELEMENT (SPEC FUNCTION)) (COLLECTION (SPEC FUNCTION)))

4 (->ATOM QUESTION2 (ELEMENT (SPEC FUNCTION)) (TEST (SPEC FUNCTION)))

5 (->CDR SYMBOL1 (FIRST-ARGUMENT FUNCTION))

6 (->RECURSE1 ACTION2 FUNCTION SYMBOL1)

7 (->T QUESTION3)

8 (->NIL ACTION3)
```

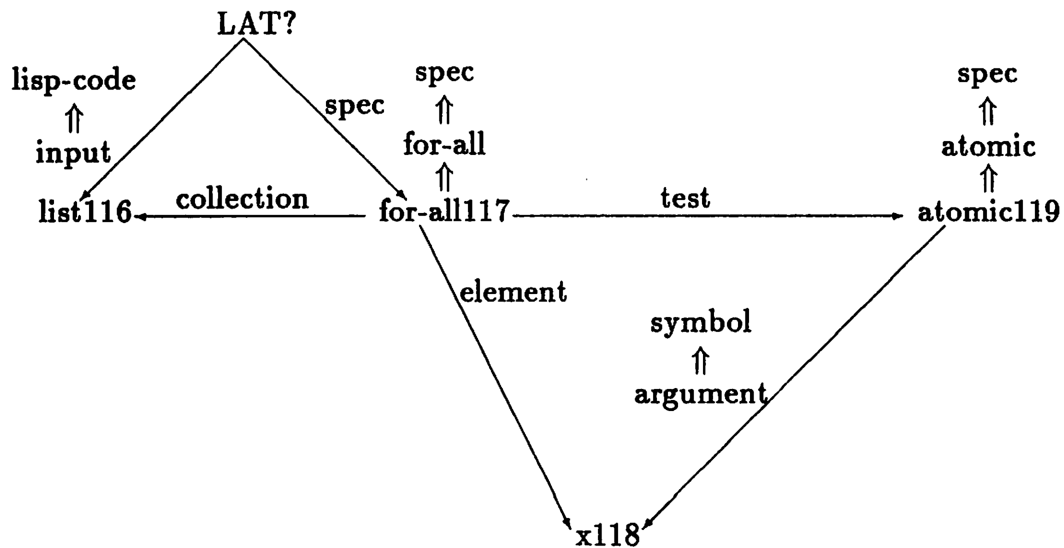

Figure 7.1: A case description.

Now, suppose that TA is asked to write the program **ALL-NON-EMPTY?**, which again takes one argument **LIST**; but which tests whether or not all elements of **LIST** are non-empty (i.e., either non-nil atoms or lists containing at least one element). This is specified to TA as (for-all x l (¬ (atomic x))) and is represented internally as in figure 7.2.

Assume that TA completes the plan for **LAT?** to match the specification for **ALL-NON-EMPTY?** by replacing step 4 in the original plan with the following step:

```
4 (->EMPTY QUESTION2 (ELEMENT (SPEC FUNCTION)) (TEST (SPEC FUNCTION)))
```

A plan schema that generalizes these two plans is created by variabalizing step 4 of the plan for **LAT?**:

```
1 (->NULL QUESTION1 (COLLECTION (SPEC FUNCTION)))

2 (->T ACTION1)

3 (->CAR (ELEMENT (SPEC FUNCTION)) (COLLECTION (SPEC FUNCTION)))

4 (* QUESTION2 (ELEMENT (SPEC FUNCTION)) (TEST (SPEC FUNCTION)))

5 (->CDR SYMBOL1 (FIRST-ARGUMENT FUNCTION))

6 (->RECURSE1 ACTION2 FUNCTION SYMBOL1)
```

[1]This assumes a linear implementation of set intersection. $x \cap y$ can be implemented in linear time by first marking all elements of x and then returning all marked elements of y.

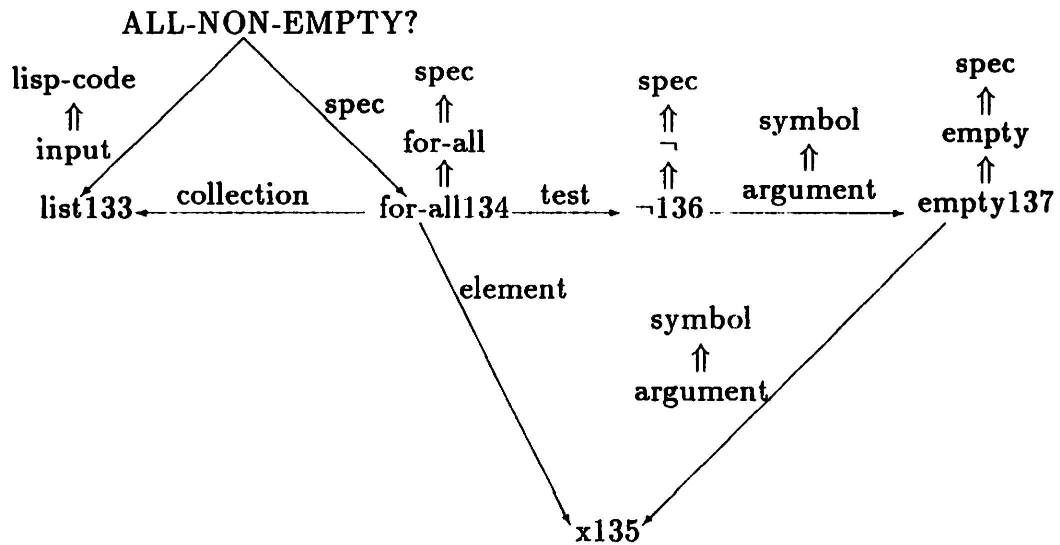

Figure 7.2: A second case description.

7 (->T QUESTION3)

8 (->NIL ACTION3)

The new schema is indexed by the generalized description shown in figure 7.3, which can be summarized as `(for-all x l (`*predicate* `x))`, where *predicate* is the most specific concept that subsumes the conflicting specification concepts. In this case, the conflicting concepts are ¬136 and ATOMIC119. These concepts have the common parent TEST, and so TEST152 is an instance of that concept. Note that the generalization process does not detect identical concepts playing multiple roles; thus, where there was one LIST concept in the specifications for LAT? and ALL-NON-EMPTY?, there are two such concepts in the general description, both with the same set of parent concepts.

7.4.3 Schema Creation and Explanation-Based Learning

It is worthwhile to compare RECODER's generalization technique with EBL techniques. RECODER is presented with the following:

- A domain theory, in the form of RECODER's prior knowledge (hierarchical relations and axiomatic plans).
- A training example, which consists of a goal and an initial state.
- A previous case, which includes its own goal, initial state, and a plan for achieving the goal from the initial state.

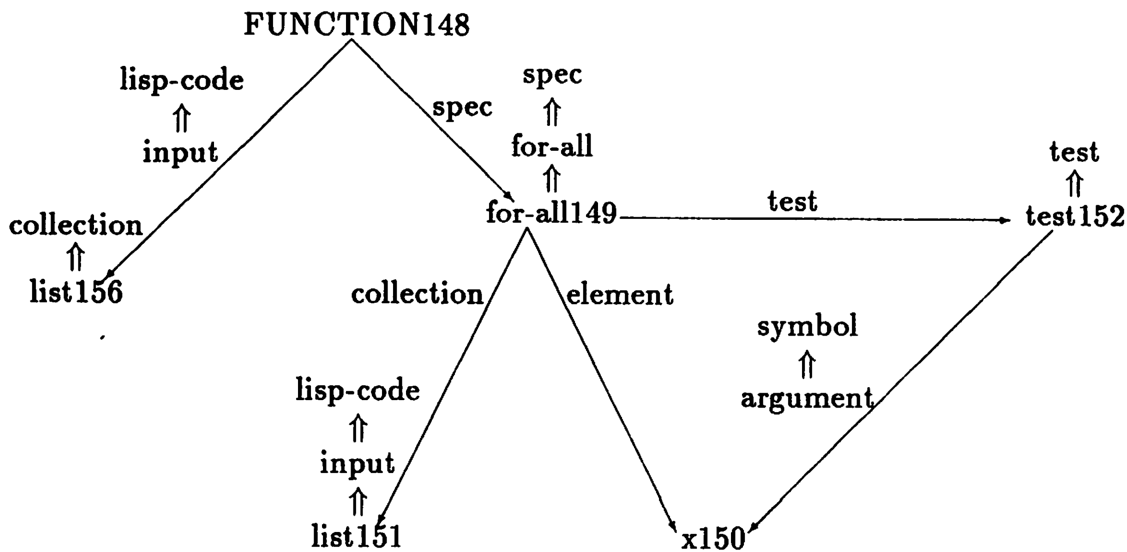

Figure 7.3: A generalized description.

RECODER's problem is to find a generalization of the example in terms of the previous case, assuming that the plan for that case can be completed to yield a plan for the example. Instead of using goal regression techniques to find a generalization of the example via an explanation (which for RECODER has its analog in a plan), RECODER creates a generalization of the example and the previous case using similarity-based techniques. On the other hand, both EBL and RECODER work from a solved problem to a generalized description of a set of concepts: in EBL the description reflects the class of initial states for which a particular goal state can be derived, while in RECODER the description reflects the class of cases that can be completed in a particular way. So, while solutions to problems play the same role in RECODER and EBL in deciding *when* generalizations are to be made, they do not play the same role in determining the *kinds* of generalizations made.

Because the kinds of generalizations made by RECODER are different from those in EBL, we see RECODER more as motivated by EBL than as extending it. For RECODER, new classes are created through a combination of its underlying case-based philosophy and the notion that generalizations should be made only if they are useful to the problem-solving process. The decision to generalize is dependent on whether an old plan can be successfully completed to form a useful plan. The details of the generalization are determined by the features of the current problem-solving situation and those of the retrieved case. This facet of RECODER represents an extension to work in concept clustering [Kolodner, 1984, Lebowitz, 1986, Fisher, 1987] in which the clustering problem is solved through appeal to task-oriented utility, rather than through appeal to heuristically based similarity metrics.

7.5 Comparison of Clustering Techniques

The point of view that we take with RECODER's learning mechanism is that forming new concepts should take into account task-oriented information. Since RECODER is a model of problem solving, this means that the results of problem solving (i.e., whether a proposed solution is successful or not) should be used in forming new concepts, whether they be concepts in the space of cases (i.e., schemas, plans, or difference maps) or concepts in the space of indices. Further, since RECODER solves problems by referring to past cases, it is appropriate to use those cases in the generalization process.

Our claim is that generalizations based on this view yields a set of concept descriptions that is more useful to a problem solver than the kinds of descriptions obtainable by traditional concept clustering methods. To support this, we have run TA on a set of 35 specifications for programs of the sort found in [Friedman, 1974]. We have used these same specifications to build generalization trees with UNIMEM [Lebowitz, 1986] and COBWEB [Fisher, 1987].

7.5.1 TA's Generalizations

TA was presented with a total of 35 program specifications. Of these 12 were presented with corresponding programs, and the rest were written by TA. TA looked first for schemas which might be appropriate, and then (if no schemas could be found) for relevant past plans. In all cases but one, the first past plan retrieved could be completed successfully (in the one case in which this was not so, TA continued searching until a plan that could be completed was found). After successful completion of plans, schemas were created by generalizing the retrieved plan with the completed plan. Descriptions used to index the 12 schemas created are summarized in figure 7.4; these descriptions use ellipses to denote concepts that were generalized.

The generated descriptions enabled TA to retrieve schemas that could be completed successfully when such schemas existed. Thus, they served to partition the space of program specifications according to their usefulness in the completion process, as would be expected from the way in which the schemas were created.

7.5.2 Generalizations from UNIMEM and COBWEB

The same set of 35 specifications used by TA was given to both UNIMEM and COBWEB. Each of these clustering algorithms was given the specifications in two forms of attribute-value pairs: one intended to capture the structured nature of specifications in TA, and one ignoring structure in its representation of attributes. For example, consider a `for-all` concept with a `test` role played by an `if-any` concept, itself with a `test` role played by an `atomic` concept. The structured

```
(for-all ...)
(if-any ...)
(first-found ...)
(if ...)
(the-first ...)
(each ...)
((deep each) ...)
((tail-append each) ...)
((tail-reverse each) ...)
((listified (deep each)) ...)
((tail-append (deep each)) ...)
((tail-reverse (deep each)) ...)
```

Figure 7.4: Concept descriptions generated by TA.

Table 7.1: Concepts found via clustering algorithms: structured features bias

	COBWEB		*UNIMEM*	
Order	*Overlap*	*Error*	*Overlap*	*Error*
Original	.167	.750	.250	.571
Permutation 1	.167	.714	.250	.667
Permutation 2	.083	.800	.250	.571
Permutation 3	.250	.625	.333	.500
Permutation 4	.167	.750	.667	.428
Permutation 5	.167	.778	.333	.600

representation of the `atomic` concept's attribute is `for-all-test-if-any-test`, while the flat representation is simply `test`.

Additionally, COBWEB and UNIMEM were each trained on six different presentation orders: the original order in which specifications were given to TA, and five permutations of that order. Results are summarized in tables 7.1 and 7.2. The second and third columns of each of these tables show overlap and error rates for COBWEB; the fourth and fifth columns show the same rates for UNIMEM. Overlap in these tables is the ratio of target concepts created to total target concepts, while error is the ratio of non-target concepts created to total created concepts. Ideally, overlap should be high and error low; as shown in these tables, the opposite was generally the case.

Figures 7.5 and 7.6 show the specific trees created with the structured feature

Table 7.2: Concepts found via clustering algorithms: flat features bias

	COBWEB		*UNIMEM*	
Order	*Overlap*	*Error*	*Overlap*	*Error*
Original	.250	.500	.917	.312
Permutation 1	.250	.625	.250	.824
Permutation 2	.333	.428	.250	.812
Permutation 3	.250	.625	.500	.625
Permutation 4	.250	.625	.333	.750
Permutation 5	.333	.500	.417	.722

bias and original presentation order for both UNIMEM and COBWEB. The tree created by UNIMEM, shown in figure 7.5, used parameter settings that allowed generalizations to be created with two features in common at the root and one additional common feature at nodes below the root, and allowed generalizations with one contradictory feature (these same parameter settings were used in all the trees created by UNIMEM). UNIMEM formed three concept descriptions that were essentially identical to ones formed by TA, but did not form the remaining nine descriptions. UNIMEM formed four concept descriptions not formed by TA. The node indexing the specification `(first-found x 1 (atomic x))` can be construed as describing specifications of the form `(... (atomic x))` (under the assumption that it describes the cases it indexes as well as the cases indexed by its children), thus adding a fifth description not formed by TA, or it could be construed as describing the single specification that it indexes.

The tree formed by COBWEB is shown in figure 7.6. It contains only two descriptions formed by TA, missing the remaining ten. It also contains six descriptions not formed by TA, as well as a node that indexes the single specification `(for-all x 1 (empty x))`.

The trees formed by UNIMEM and COBWEB have more in common with each other than with the collection of descriptions formed by TA. Each contains the three descriptions `(... (insert-left ...))`, `(... (insert-right ...))`, and `(... (replace ...))`, although COBWEB further subdivides these descriptions as shown in figure 7.7. Each also contains the concepts `(if ...)`, `(if-any ...)`, and `(... (¬ (atomic x)))`, for a total of six descriptions in common. Interpreting the non-leaf node that indexes a case in UNIMEM as describing only the case that it indexes (i.e., `(first-found x 1 (atomic x))`), UNIMEM contains only two descriptions not formed by COBWEB, and COBWEB contains only three descriptions not formed by UNIMEM. Of the 35 specifications used in forming descriptions, only five are indexed differently in COBWEB than they are in UNIMEM.

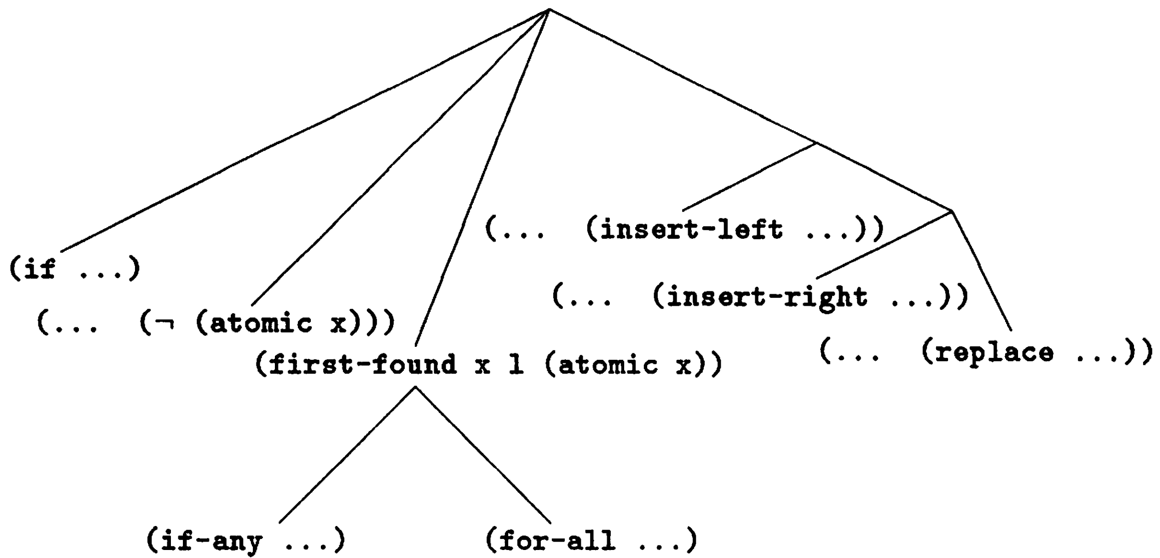

Figure 7.5: UNIMEM tree for program specifications.

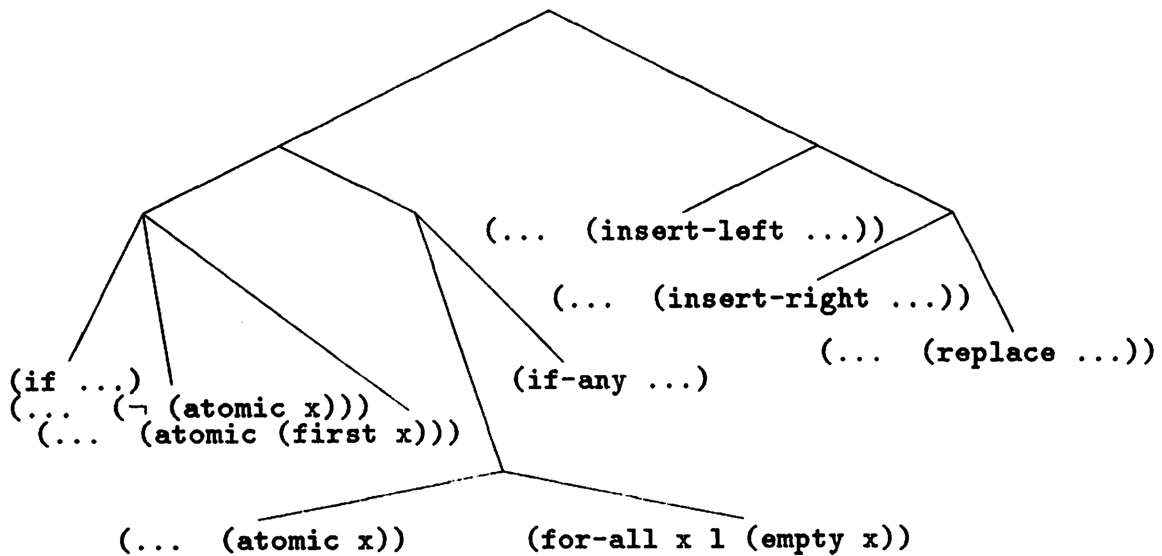

Figure 7.6: COBWEB tree for program specifications.

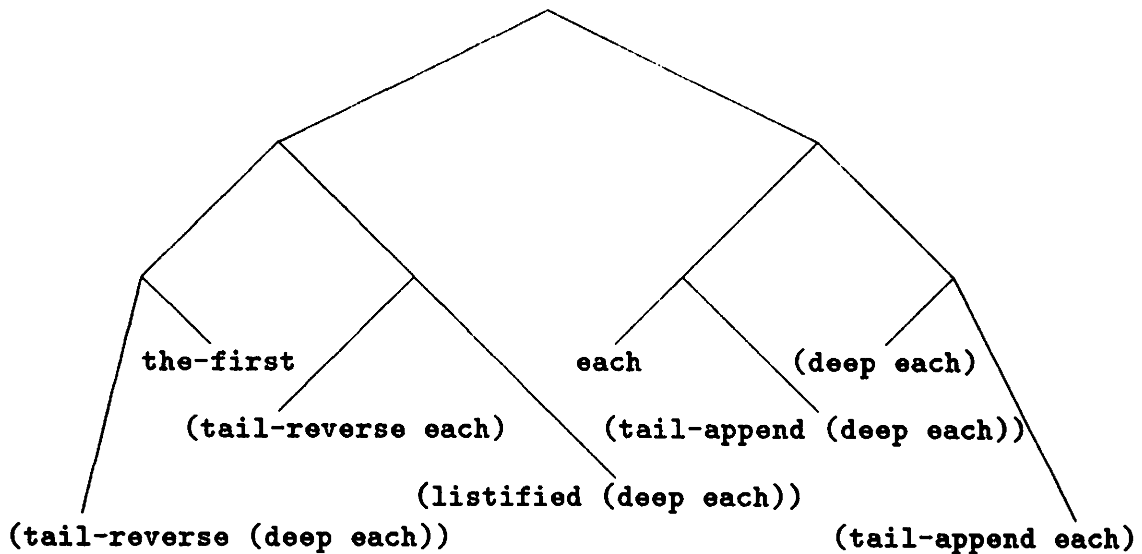

Figure 7.7: Further subdivisions of the COBWEB tree.

Table 7.3: Specifications and corresponding program names.

Specification	*Program*
`(for-all x l (atomic x))`	`LAT?`
`(for-all x l (empty x))`	`ALL-EMPTY?`
`(for-all x l (atomic (first x)))`	`ALL-ATOM-CAR?`
`(if-any x l (atomic (first x)))`	`HAS-ATOM-CAR?`
`(first-found x l (atomic x))`	`FIRST-ATOM`

These five are shown with their corresponding program names in table 7.3. The differences in indexing are shown in table 7.4.

The results from UNIMEM and COBWEB show a significant shortcoming of clustering methods that only take similarity of surface features into account. These techniques can create clusters that are not useful in the task for which they are intended to be used. In the case of the programming domain, none of the clusters created by UNIMEM or COBWEB that indexed more than one case and that were not also created by TA is useful in writing programs. Each one of those clusters abstracts out concepts in such a way that subsequent successful completion is unlikely.

For example, the `(... (¬ (atomic x)))` description in both COBWEB and UNIMEM indexes the two specifications `(if-any x l (¬ (atomic x)))` and

Table 7.4: Differences in categorization between COBWEB and UNIMEM.

Program	*Category in UNIMEM*	*Category in COBWEB*
LAT?	(for-all ...)	(... (atomic x))
ALL-EMPTY?	(for-all ...)	(for-all x 1 (empty x))
ALL-ATOM-CAR?	(for-all ...)	(... (atomic (first x)))
HAS-ATOM-CAR?	(if-any ...)	(... (atomic (first x)))
FIRST-ATOM	(first-found x 1 (atomic x))	(... (atomic x))

(first-found x 1 (¬ (atomic x))). A program schema that captures the similarities of these two specifications is

```
(cond ((null list) nil)
      ((not (atom (car list))) lisp-code)
      (t (function (cdr list))))
```

Suppose that a program is to be written for the specification (for-all x 1 (¬ (atomic x))). Any attempt to complete something like the above program schema would fail, since more needs to be changed than the items that were generalized. For a generalization to be successfully completed, it would need to take the form:

```
(cond ((null list) lisp-code)
      ((not (atom (car list))) lisp-code)
      (t lisp-code))
```

Unfortunately, no technique that uses similarity-based generalization will produce the above schema, since it overgeneralizes unnecessarily.

On the other hand, one could argue that generalized specification descriptions should not be used to index plan *schemas*, but that they should rather index the plans themselves. But even if one or the other of the original plans described by (... (¬ (atomic x))) were used, the completion attempt would fail, since there are no axiomatic plans in TA for dealing with the FOR-ALL specification concept.

Here, one could argue that this is a failing of the particular set of axiomatic plans employed by TA — if a different set of plans were used, a set that more accurately captured the nature of the generalizations produced by either COBWEB or UNIMEM, problem-solving performance would be improved. Unfortunately, this method of producing axiomatic plans would require updating the set of axiomatic plans whenever the clustering algorithm being used produced a new generalization. Alternately, the approach would require a set of axiomatic plans that reflected

perfect knowledge of the domain. Neither of these alternatives seems very appealing. The first alternative demands that the problem solver be driven by the syntactic features of the problems to be solved; the second demands that all problem-solving knowledge be available before any problems can be solved.

7.6 Conclusions

We have presented a technique for creating and describing plan schemas, called analytic concept creation, that takes advantage of the case-based characteristics of case-based problem solving as well as its problem-solving characteristics.

The case-based nature of the endeavor allows as to think of solving problems as modifying old solutions to fit new situations. After modification takes place we are left with an old solution and a new solution, and can generalize both the solutions and the situations in which the solutions apply using techniques related to concept clustering.

The problem-solving nature of case-based problem solving allows us to focus the application of clustering techniques. Rather than attempting to create clusters based on feature similarity alone, we use the successful modification of past solutions as a guide for creating clusters. Thus, as in explanation-based learning, the problem-solving process is used to guide the generalization process.

We have shown, by comparing analytic concept creation to more traditional clustering approaches, that this technique yields clusters that are more useful in future problem solving than those techniques that use clustering metrics based solely on feature similarity. This suggests that in problem-solving domains, analytic concept creation is a viable technique for creating generalizations related to the act of problem solving.

This does not, of course, diminish the role of clustering techniques that are based solely on feature similarity. These techniques can be quite useful in knowledge-poor domains, and for tasks (such as prediction of features, as in IPP [Lebowitz, 1983]) unrelated to problem solving per se.

Chapter 8

Completing the Model

8.1 Introduction

This thesis has concentrated on examining the role of generalizations in case-based problem solving. A model of case-based problem solving, called RECODER, was introduced, and through that model certain questions about the appropriateness and utility of certain types of generalization have been addressed.

In particular, the work in this thesis makes three major claims:

1. It is not necessary to generalize descriptions of cases in order to index cases in memory. An approach that uses a parallel partial-matching algorithm, indexing on complete descriptions, will retrieve at least as well.

2. There are important uses for generalization in case-based problem solving. In particular, they can make problem solving more efficient, and they can be useful in problem diagnosis.

3. A successful technique for generalizing cases can be developed that uses the strengths of both the case-based and problem-solving aspects of case-based problem solving.

These claims have been demonstrated with respect to RECODER in the body of the thesis.

RECODER was originally conceived as a general model of case-based problem solving. While this thesis has focused on the question of generalization within that model, this does not imply that other areas of the model have no need of further study. In particular, the problem-solving and debugging components of the model need to be further examined.

There are three remaining sections to this conclusion. The first will review the current state of the art of case based problem solving, broken down according to the components of RECODER. The following section will compare the work described herein with a previous version of the TA program-writing system, pointing up differences in approach. Some of the ideas in the original version of TA that were left out of the current version show up again in the final section, which looks at possible directions for future work.

8.2 State of the Art

8.2.1 Retrieval

Much of the work in case retrieval is descended from Schank's work in dynamic memory [Schank, 1982]. Implementations of Schank's theories of memory organization can be found in CYRUS and UNIMEM [Kolodner, 1984, Lebowitz, 1986], and in more recent systems such as the PERSUADER [Sycara, 1987]. The important aspect of this line of work, at least from the point of view of case-based reasoning, is really the *selection* of features to be used in the storage of and search for cases, rather than the actual techniques to be used in organizing memory. Other recent work in selecting indices uses techniques from explanation-based learning, and is exemplified by [Barletta and Mark, 1988].

That some selection process is necessary to derive useful descriptions of cases will not be argued here; indeed TA uses its own selection criteria, indexing different types of cases in different ways. However, we feel that it is important to draw a distinction between selection of features to be used for retrieval and the retrieval process itself. We agree with Waltz [Waltz, 1989] that retrieval itself should be fast and simple. Thus, the emphasis in RECODER has been on a memory organization built around a parallel retrieval algorithm. As we have shown in this thesis, such an algorithm can compare favorably to an algorithm based on searching a dynamically constructed discrimination net, as in CYRUS, UNIMEM, and similar systems.

8.2.2 Completion

There are several approaches to modifying past solutions of problems to fit current situations. In analogical transformation [Carbonell, 1983], solutions are treated as declarative objects to which transformations are applied. Abstractional analogy [Shinn, 1988] also treats past solutions as declarative objects, but creates new plans by abstracting out the differences in the problem situations to create an abstract solution; this is then instantiated with reference to the current situation. Derivational analogy [Carbonell, 1986] takes into account the problem-solving process itself, following the process used for the old solution and making modifications along the way. Adaptive planning [Alterman, 1988] attempts to execute the old plan, adapting to the current situation on the fly.

RECODER's completion component is most similar to adaptive planning, although it is more cautious. Rather than actually executing an old plan, RECODER merely simulates execution. Additionally, RECODER does not automatically assume that adaptation will be necessary; rather, replanning is only carried out on a plan step if that step refers to an inconsistency between the initial situation of the old plan and the current situation.

8.2.3 *Debugging*

Interestingly, adaptive planning [Alterman, 1988] serves as an example of both completion and debugging, the latter because its adaptations are generally precipitated by plan failures. Other examples of debugging failed (or potentially failed) plans are CHEF [Hammond, 1986a] and KIP [Luria, 1988]. Each of these share an extensive amount of knowledge for analyzing failed plans to determine why they failed (although the types of knowledge differ). This is in stark contrast to debugging in RECODER, which is dependent (either directly or indirectly) on advice from an experienced user. This is both advantageous and disadvantageous. It is an advantage because a major knowledge engineering effort is saved. It is not necessary to write debugging rules; rather, debugging knowledge is acquired in the form of cases through the process of making mistakes. It is a disadvantage because not even the simplest of bugs can be analyzed in the simplest of ways by RECODER in its present form. What is required is a way to combine the advantages of reasoning by cases with those of reasoning by other forms of knowledge; this will be discussed in the section on future work.

8.2.4 *Reorganization*

The consensus among much of the case-based reasoning community is that CBR systems capable of dynamically acquiring new cases are by that capability alone learning systems [Plan, 1989]. Consequently, "learning in CBR systems" has come to be almost synonyms with "acquiring new cases in CBR systems." Since CBR systems can indeed improve their performance by adding new cases, we agree that this does qualify such CBR systems as learning systems. However, we believe that there is more to learning in CBR systems than simply acquiring new cases. A major part of this thesis has argued that there are important roles that can and should be played by generalizations of cases. We are aware of little other work in CBR that has made such arguments, although both Carbonell [Carbonell, 1983] and Shin [Shinn, 1988] share our interest in generalizing cases.

8.3 *A Brief History of TA*

The original TA system, which I will here call TA.1, was primarily a case-based debugging system. As with the current version (here called TA.2) it was primarily concerned with writing LISP programs of the style found in the Little Lisper [Friedman, 1974], but there are many important differences between the two systems. This section will describe TA.1 and compare it to TA.2, in the hopes that such a comparison will reflect not only what has been accomplished with the RECODER model, but also some problems that need to be solved. In particular

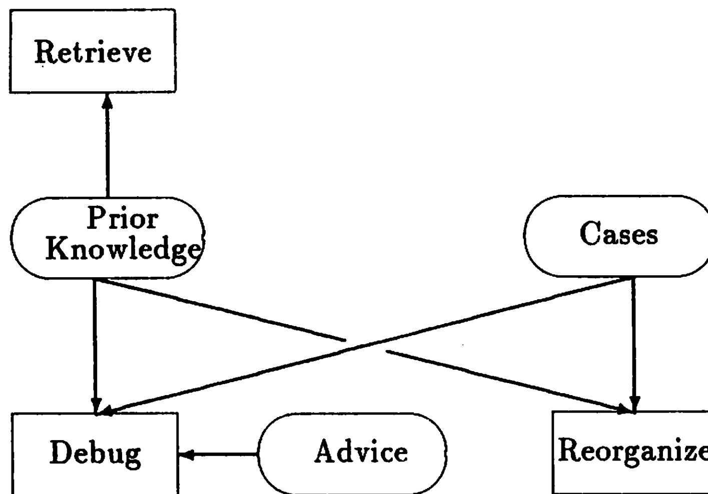

Figure 8.1: The architecture for TA.1.

we will see that, while the completion component of RECODER, as implemented in TA.2, provides capabilities not available in TA.1, TA.1 provides capabilities in the area of debugging not provided in TA.2. Incorporation of these capabilities into RECODER may yield a more comprehensive model of debugging.

8.3.1 TA.1

Superficially, TA.1 [Williams, 1988a, Williams, 1988b] is quite similar to TA.2. At the architectural level, TA.1 can be seen as implementing the RECODER model without the completion component, as illustrated in figure 8.1 (for ease of comparison, figure 8.2 reviews the RECODER model).

A closer examination of the knowledge sources available to TA.1, though, uncovers certain differences.

1. *Cases.* The primary type of case in TA.1 is a program. In TA.2, which implements the RECODER model, the primary type of case is a plan which, when executed, will create a program. Rather than an implementation of a general-purpose planning model, TA.1 is strictly a model of programming.

2. *Prior knowledge.* TA.1 has only one type of prior knowledge available to it: what I shall here refer to as a *naive model* of programming.

3. *Advice.* The types of advice needed by the two versions of TA differ in certain ways, mostly due to the differences in the types of cases and prior knowledge available to the two systems.

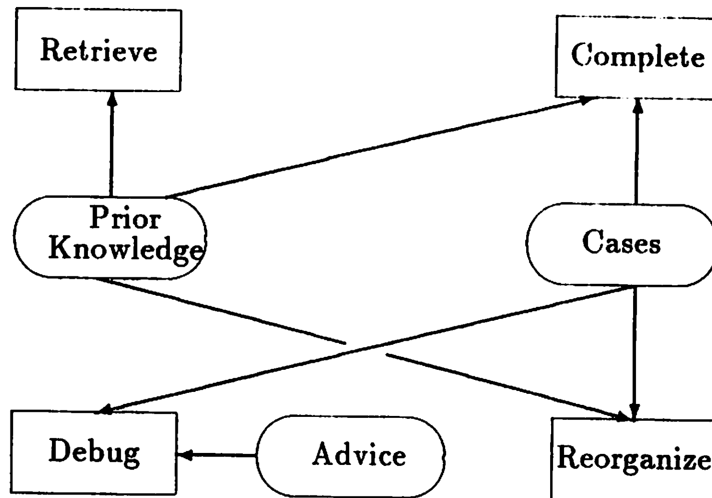

Figure 8.2: The architecture for TA.2 (i.e., RECODER).

The differences in available knowledge affect each of the three major components of TA.1, as will be shown below.

8.3.2 Retrieval

Figure 8.3 shows the organization of TA.1's retrieval component. The retrieval itself is done using the same process as that described for RECODER (and used in TA.2); the difference is in the way cases are indexed. TA.2 (as described elsewhere) uses internal representations of specifications, while TA.1 uses its naive model.

TA.1's naive model (as described in [Williams, 1988a]) is intended to represent the way in which naive programmers might think about programs. In particular, most programs that involve iteration (or, in LISP, recursion) are represented in the naive model as searches through a space of the objects being iterated. As an illustration, following is the naive program for the specification `(forall x l (atomic x))`:

```
(SEARCH (SEARCH-SPACE-TYPE CONS)
        (SEARCH-SPACE L)
        (SEARCH-TYPE LIST-SEARCH)
        (SEARCH-ELEMENT X)
        (STOP-AFTER ALL-SEARCHED)
        (ON-TERMINATION SUCCEED)
        (TEST
          (TESTER (TEST (ISA X ATOM))
                  (ON-SUCCESS SUCCEED)
```

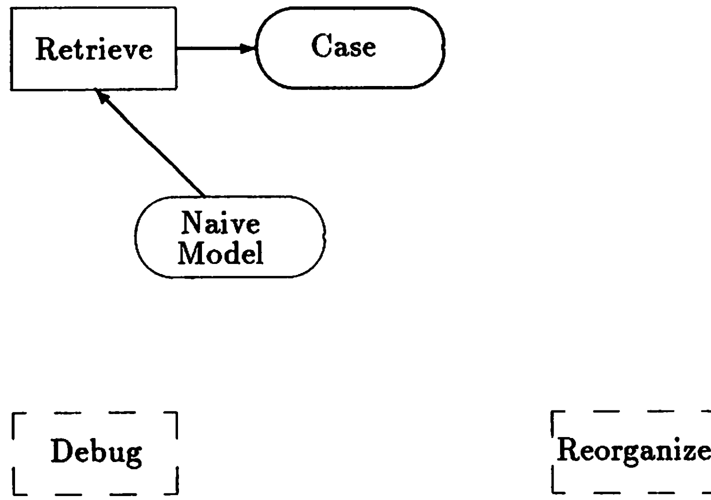

Figure 8.3: TA.1's retrieval component.

```
            (ON-FAIL FAIL)))
    (ON-SUCCESS CONTINUE)
    (ON-FAIL FAIL))
```

This can be thought of as a SEARCH frame that contains several slots: SEARCH-SPACE-TYPE, SEARCH-SPACE, SEARCH-TYPE, etc. It describes a search in a space consisting of a CONS object referred to as L. The type of search is a list search, with X denoting the current search element. The search will stop, barring failure, after the space has been exhausted; if this happens, it constitutes a successful search. The test to be applied to each search element is a test for atomicity; if successful, the search will continue; otherwise the search will end with failure.

8.3.3 Debugging

The only way that old programs can be modified in TA.1 is through its debugging component, whose organization is shown in figure 8.4. This debugging component differs from that in TA.2 in several respects, and it has some capabilities that are not present in TA.2.

The overall debugging process in TA.1 is similar to that in TA.2. A sample program call is given to the system; if the call does not produce the correct result, prior cases that failed in similar ways are searched for. If none are found, interactive tracing is undertaken. TA.1 does separate what in TA.2 are called difference maps into two kinds of cases: *bugs* and *patches*. Bugs are indexed by similarities between cases, and refer to the parts of buggy programs that actually contained the bugs. Patches are indexed by differences between cases, and contain transformations from

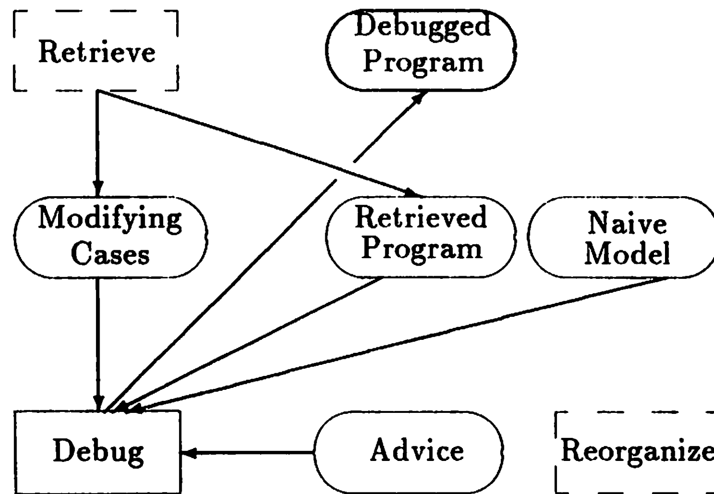

Figure 8.4: TA.1's debugging component.

incorrect program fragments to correct ones.

Certain decisions in tracing buggy programs that must be made by the human user in TA.2 were made by the system itself in TA.1, due largely to the capabilities of the naive model. The model allowed for a dual interpretation of a specification: a naive program can be thought of not only as a sequence of symbols, useful for indexing cases, etc., but also as a means by which to emulate the desired performance of a program. This latter was achieved by providing a kind of naive program interpreter, which could carry out the searches that were represented by the model. This allowed TA.1 to decide for itself whether or not a given program did what it was supposed to do, without having to ask for advice.

Additionally, TA.1 allows connections to be made between fragments of naive programs and fragments of LISP programs. For instance, TA.1 might learn that the naive program fragment `SUCCEED` is represented by the LISP fragment `T`. These connections are explored in interactive sessions tracing correct programs, and are remembered in two case bases, one that indexes LISP fragments by naive fragments (these are called *fragments*), and one that indexes naive fragments by the positions of their corresponding fragments in LISP programs (these are called *purposes*).

8.3.4 Reorganization

Figure 8.5 shows TA.1's reorganization component. TA.1 does not create generalizations, but merely adds new cases, indexed appropriately. The primary function of the reorganization component is to determine a set of features to index a new case by. Indices for each of the types of cases are as follows:

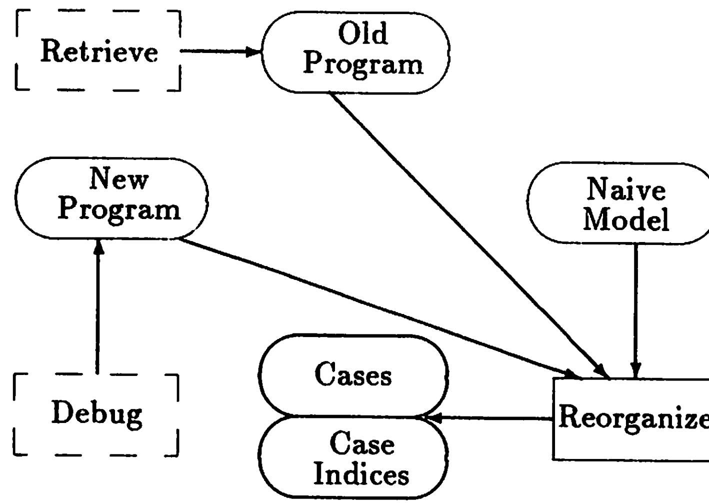

Figure 8.5: TA.1's reorganization component.

- Programs are indexed by their naive representations.
- Program fragments are indexed by the corresponding fragments of their naive representations.
- Purposes (i.e., fragments of naive programs) are indexed by their position in the corresponding program (e.g., question of first COND clause, etc.).
- Bugs are indexed by the similarities between the naive representations of the programs containing bugs and the corrected versions of those programs.
- Patches are indexed by the difference between the naive representations of the programs needing to be patched and the patched versions of those programs.

8.3.5 *Comparing TA.1 and TA.2*

To compare the capabilities of TA.1 and TA.2, we will go through a small program-writing session, showing how each would work on the same set of programs.

To begin, let us assume that each has the program LAT? in its case base. LAT? is specified as `(forall x l (atomic x))` and its associated LISP program is

```
(cond ((null l) t)
      ((atom (car l)) (lat? (cdr l)))
      (t nil))
```

Table 8.1: Associations between LISP code and naive model.

Fragment part	*LISP fragment*	*Naive fragment*
Question 1	`(null l)`	ON-TERMINATION
Action 1	`t`	ON-TERMINATION
Question 2	`(atom (car l))`	TEST
Action 2	`(lat? (cdr l))`	ON-SUCCESS
Question 3	`t`	ON-FAIL
Action 3	`nil`	ON-FAIL

In addition, let us assume that in TA.1, we trace some calls of LAT?, associating program fragments with fragments of the naive version of the program specification. That version, listed earlier, is repeated here for convenience:

```
(SEARCH (SEARCH-SPACE-TYPE CONS)
        (SEARCH-SPACE L)
        (SEARCH-TYPE LIST-SEARCH)
        (SEARCH-ELEMENT X)
        (STOP-AFTER ALL-SEARCHED)
        (ON-TERMINATION SUCCEED)
        (TEST
          (TESTER (TEST (ISA X ATOM))
                  (ON-SUCCESS SUCCEED)
                  (ON-FAIL FAIL)))
        (ON-SUCCESS CONTINUE)
        (ON-FAIL FAIL))
```

Tracing yields the associations shown in table 8.1. The table shows only slots of naive fragments; these are what are stored in the purpose case base. The values of the fragments are used to index LISP fragments, e.g. the fragment `(lat? (cdr l))` is indexed by CONTINUE, the value of the ON-SUCCESS slot.[1]

Now, let us suppose that both TA.1 and TA.2 are asked to write the program HAS-ATOM?, specified as `(if-any x l (atomic x))`. TA.1 will create the following naive model for this specification:

```
(SEARCH (SEARCH-SPACE-TYPE CONS)
        (SEARCH-SPACE L)
```

[1]LISP fragments are additionally indexed by their function in the program; e.g., whether they are questions or actions in COND clauses.

```
(SEARCH-TYPE LIST-SEARCH)
(SEARCH-ELEMENT X)
(STOP-AFTER ALL-SEARCHED)
(ON-TERMINATION FAIL)
(TEST
  (TESTER (TEST (ISA X ATOM))
          (ON-SUCCESS SUCCEED)
          (ON-FAIL FAIL)))
(ON-SUCCESS SUCCEED)
(ON-FAIL CONTINUE))
```

Both versions of TA will retrieve the only program available to them, namely LAT?. Both version will have to debug this program, since TA.2 will be unable to complete it and TA.1 has no completion component. Each version will need to debug three separate calls to the buggy program, because three separate parts of LAT? need to be altered to produce a correct program. But for each call, TA.1 needs to rely on advice from the user to a much lesser degree than does TA.2.

For a given call, TA.2 must ask for advice from the user regarding the expected outcome of the call. TA.1 is able to use its naive model to determine what the outcome should be. Once a fragment of code has been found whose result differs from the expected result, TA.2 must ask for advice about what the correct code should be. TA.1, though, can get the information it needs from the associations made during tracing. For instance, consider the action in the third COND clause of LAT?. TA.1 knows from its purpose case base that this action is related to the ON-FAIL part of its naive program. The value of this part is CONTINUE. CONTINUE indexes a recursive call in the fragments case base (by the associations in table 8.1), and so the action of the third COND clause in HAS-ATOM? becomes a recursive call.

There are times when TA.1 does not perform as well as TA.2. For instance, consider the program ALL-NON-ATOMS?, specified as (forall x l (¬ (atomic x))). TA.1 creates the following naive version for that specification:

```
(SEARCH (SEARCH-SPACE-TYPE CONS)
        (SEARCH-SPACE L)
        (SEARCH-TYPE LIST-SEARCH)
        (SEARCH-ELEMENT X)
        (STOP-AFTER ALL-SEARCHED)
        (ON-TERMINATION SUCCEED)
        (TEST
          (TESTER (TEST (ISA X ATOM))
                  (ON-SUCCESS FAIL)
                  (ON-FAIL SUCCEED)))
        (ON-SUCCESS CONTINUE)
        (ON-FAIL FAIL))
```

Both versions of TA will again retrieve LAT?. TA.2 easily creates a correct program via its completion component, using axiomatic plans for $\neg$ and ATOMIC. TA.1, not having a completion component, must attempt to debug LAT?. However, not having seen a TEST slot (or any naive fragment) with a value the same as that in ALL-NON-ATOMS?, TA.1 must resort to asking the user for the correct code.

In a way, the naive model (augmented by interactive tracing of correct programs) provides a complement to the axiomatic plans available to RECODER's completion component. Associations between naive programs and LISP programs allow TA.1 to discover certain types of program schemas without resorting to interactive debugging. This is essentially what happened in writing HAS-ANY?. On the other hand, axiomatic plans provide a convenient way of making local modifications to plans without altering their schematic properties, as evidenced by TA.2's successful completion of the plan for ALL-NON-ATOMS?.

8.4 *Further Work*

There are three areas in which work on RECODER can continue:

1. More robust implementation.

2. Examination of the role of planning in case-based problem solving, via RECODER's planning component.

3. Examination of the role of debugging in case-based problem solving, via RECODER's debugging component.

The implementation area is mostly an engineering issue, but it is important nonetheless. The retrieval component as implemented is somewhat inefficient, which tends to frustrate extensive testing. More importantly, though, the planning component is implemented in a somewhat ad hoc fashion, with heuristics hardcoded in. This tends to limit the kinds of plans that can be written to those plans on which the hardcoded heuristics happen to work; moreover, it is a time-consuming process to modify the heuristics. A better implementation would layer a traditional search-based planner with heuristics that are easy to add and modify. This would allow a wider range of plans to be generated, and would thus facilitate analysis of the planning component.

There are several types of analysis that could be performed on the planner, both in the domain of programming and in other domains. Some of these include:

- Comparing planning via completion of cases to planning from scratch. Both the planning effort required and the prior knowledge needed could be compared.

- Further examining the utility of schemas, e.g. in other planning domains and with respect to both their efficiency in planning and their use as diagnostic or explanatory aids.

- Examining the utility of chunks, comparing time spent searching for chunks with time spent in using chunks (as opposed to generating plans). This type of effort is currently weakened by the implementation of chunks, which rests on the frailties of the planner.

There are many questions that need to be addressed with regard to RECODER's debugging component; it is currently the least-developed part of the model. The most promising aspect of debugging, that of using plan schemas to help diagnose buggy plans, does not even arise from the debugging component, but from the completion and reorganization components. Further, While schemas do help in diagnosis, by providing context for bugs, it is not clear how relevant schemas are to be chosen. One can imagine heuristics for selecting schemas, e.g., choosing the schema most similar to the buggy plan. It would be worthwhile to explore this line of work in an interactive debugging environment.

The debugging component can be used for diagnosis; for example, it is possible to diagnose a certain class of bugs using difference maps. But the kinds of permutations captured by difference maps do not tend to occur in the programming protocols analyzed earlier. What actually seems to be causing the interesting bugs is a confusion resulting from attempting to combine incompatible schemas — e.g., trying to combine a tail-recursion schema with a recursion schema for recursing on both the **car** and the **cdr** of a list. Other bugs are caused by extrapolation of identity — that is, two variables are interchangeable in one situation, and are mistakenly thought to be interchangeable in another. While some bugs may arise from simply using an incompleteable plan, these don't — rather, they arise from completing a plan (or schema) *incorrectly*. The current model doesn't cover this.

In addition to bug diagnosis, bug repair needs to be examined more fully. One way to handle the repair question is with an avoidance strategy, as in CHEF. This is entirely reasonable; in fact, it could be argued that by choosing past plans with as few differences from the current situation as possible, one is avoiding possible failures caused by plans with many differences. Sometimes, however, avoidance is not enough; even CHEF must be equipped with the ability to detect and repair bugs. In fact, CHEF caches repairs in a way analogous to RECODER [Hammond, 1986d]. It is not clear, though, that the specific structure employed by RECODER, i.e. the difference map, is the best approach to repairing plans. Another approach would be to use differences to index not difference maps, but rather debugged plans. Then, when a plan has failed, RECODER could find a plan that fixed a similar failure, and attempt completion on that new plan.

Finally, RECODER's approach to bug detection needs to be examined more fully. It is entirely unrealistic to demand that detection can only be done with

the aid of an external oracle, as is currently the case. Novice programmers do not need to ask whether their program has produced the result that it was supposed to; they usually can figure out what the correct answer is *supposed* to be, even if they cannot write a program to compute it. This is where the naive model used by TA.1 would be quite useful. Such a model would allow TA to do a "naive computation" to determine what a program *should* do. In addition, the naive model allows connections to be built up between "naive programs" and actual programs, through supervised tracing of correct programs.

The naive model was abandoned in the transition from TA.1 to RECODER primarily in an effort to give focus to the work — the naive model was incomplete and its implementation was fragile, and it didn't fit with the direction in which RECODER was heading at the time. Further, the model as it existed in TA.1 was limited to reasoning about programming. But the model, at least with respect to the programming domain, promises to fill in some gaps in the current version of RECODER, and provides a kind of prior knowledge that nicely complements that of axiomatic plans. It is not yet clear whether a general naive model of planning can be developed, but if it can, it will certainly enrich the capabilities of RECODER.

APPENDICES

Appendix A

Completion Time Comparison Specifications

Following are the specifications for the 11 programs generated in the experiment comparing completion with and without schemas. Each specification contains a *form* (the form of a call to the program, i.e., a list containing the program's name followed by its arguments), a *paraphrase*, (an English paraphrase of what the program is to do), and a *spec* (the specification as presented to RECODER).

1. *form:* (has-empty? ll)
   paraphrase: Does `ll` contain an empty list?
   spec: (if-any x ll (empty x))

2. *form:* (all-atom-car? ll)
   paraphrase: Is the first element in each sublist of `ll` an atom?
   spec: (for-all x ll (atomic (first x)))

3. *form:* (has-atom-car? ll)
   paraphrase: Does `ll` contain a sublist whose first element is an atom?
   spec: (if-any x ll (atomic (first x)))

4. *form:* (subst o n l)
   paraphrase: Return `l`, with the first top-level occurrence of `o` replaced by `n`.
   spec: (the-first x l (same o x) (replace n x (remainder l)))

5. *form:* (multisubst o n l)
   paraphrase: Return `l`, with each top-level occurrence of `o` replaced by `n`.
   spec: (each x l (same o x) (replace n x (remainder l)))

6. *form:* (subst* o n l)
   paraphrase: Return `l`, with all occurrences of `o` replaced by `n`.
   spec: ((deep each) x l (same o x) (replace n x (remainder l)))

7. *form:* (subst*-1 o n l)
   paraphrase: Same as `subst*`, but treat `l` as a list rather than a tree.
   spec: ((listified (deep each)) x l (same o x) (replace n x (remainder l)))

8. *form:* (subst-t1 o n l r)
   paraphrase: Same as `multisubst`, but tail-recursive using **append**.
   spec: ((tail-append each) x l (same o x) (replace n x (remainder l)))

9. *form:* (subst-t2 o n l r)
   paraphrase: Same as `multisubst`, but tail-recursive using `reverse`.
   spec: ((tail-reverse each) x l (same o x) (replace n x (remainder l)))

10. *form:* (subst*-t1 o n l r)
    paraphrase: Same as `subst*-1`, but tail-recursive using **append**.
    spec: ((tail-append (deep each)) x l (same o x) (replace n x (remainder l)))

11. *form:* (subst*-t2 o n l r)
    paraphrase: Same as `subst*-1`, but tail-recursive using `reverse`.
    spec: ((tail-reverse (deep each)) x l (same o x) (replace n x (remainder l)))

Appendix B

Retrieval in RECODER and UNIMEM

B.1 Introduction

This appendix describes an experiment comparing storage and retrieval capabilities of RECODER to those of UNIMEM, a system that organizes cases in a hierarchy of generalizations created using concept clustering techniques. There were two main criteria in designing the experiment:

1. The case base should be large. Several dozen cases are not enough to give a realistic evaluation. An exponential growth rate may not be recognizable as such until several hundred cases have been processed.

2. The experiment should isolate the retrieval process. Detailed case analysis should be avoided.

To meet these criteria, we chose a domain in which many cases are available, and a task for which detailed analysis is not necessary. The domain we chose is English words, and the task is finding words with similar spellings.

The experiment has two modes: *training mode* and *testing mode*. In training mode, new words are introduced and assimilated into a case base. In testing mode, words are presented and matches are retrieved from the case base.

There are three criteria used for evaluating results:

1. The memory used in assimilating training examples. The less memory used, the better. In addition to the size of memory after training, the rate of growth of memory can be significant.

2. The relevance of retrieved cases during testing. This is a relative measure. There may be no close matches in memory for a given test example, but retrieval should still return among the best available matches.

3. The efficiency of the retrieval process. The faster the retrieval, the better.

In evaluating the similarity of retrieved words to test words, we use a metric that takes into account the closeness of the retrieved word to the test word and vice versa. The metric, $S(r,t)$ (for similarity of retrieved word r to test word t), is

$$S(r,t) \equiv \frac{2M(r,t)}{F(r)+F(t)}, \qquad \text{(B.1)}$$

where $M(x, y)$ refers to the number of features in word x for which there are identical features with identical values in word y, and $F(x)$ refers to the total number of features in word x.

The experiment will be discussed in two phases. In the initial phase, words from three to seven letters in length were used in training, and only one feature bias was considered. In the second phase, the effect on generalization of words of different lengths was minimalized by training only on words words four and five letters long. Additionally, the second phase considered different feature biases and different orderings of the training set. Finally, the first phase of the experiment involved comparing only memory size and retrieval accuracy; the second phase compared retrieval efficiency as well.

B.2 Phase One

In the first phase of the experiment, features were extracted from words (both for search and storage) as follows. Each letter of the alphabet was associated with one feature for each number from 0 up to a predefined maximum word length. Each such feature $< x, y >$ refers to the letter in a word that appears y positions after a given letter x (this feature is also referred to herein as "y-AFTER-x"). The values these features can take on are the letters of the alphabet, and each word is assigned a set of features corresponding to the set of letters following each letter in the word. For example, the word "cat" is represented by the following set of six features:

{(<C 0> C) (<C 1> A) (<C 2> T) (<A 0> A) (<A 1> T) (<T 0> T)}

The experiment's first phase involved doing separate training runs on six groups of words.[1] These groups, referred to as sets 1, 2, 3, 4, 5, and 6, contain 244, 246, 245, 253, 253, and 246 words respectively. The words are from three to seven letters in length. For testing, two groups of words were used. The first, called FAM, contains 99 words, all of which also appear in the first training group. The second training group, called UNFAM, contains 99 unfamiliar (i.e. not present in any test group) four and five letter words.

As discussed in chapter 5, the size of RECODER's memory is strictly proportional to the number of cases contained in memory. The rate of growth of RECODER memory is strictly linear. The generalization hierarchies created by UNIMEM are dependent on the values of several parameters, as well as on the order of presentation, so each group of training words was used to create more than one hierarchy. In particular, two parameters were varied: one denoting the number of common features needed to consider making a generalization (called *GENERALIZE-NUMBER*), and one denoting the number of contradictions to

[1]These words are the same words used in training and testing PRO [Lehnert, 1987a]. Unlike PRO, which was designed for word pronunciation, we were interested only in retrieving similar words.

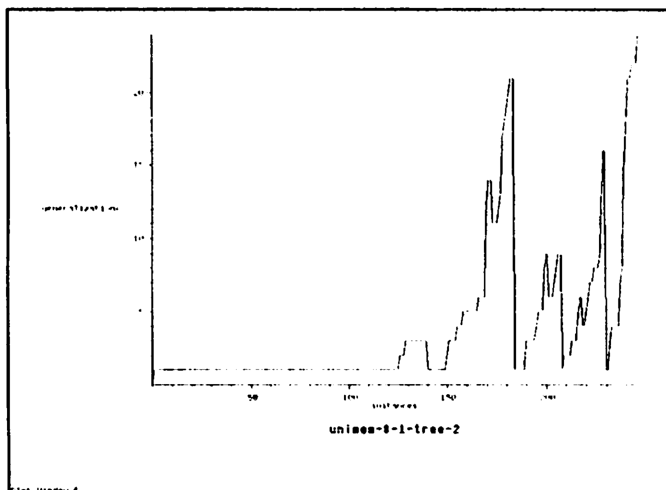

Figure B.1: History for growth of UNIMEM tree {8, 1}, trained on set 2.

be tolerated in a generalization (called *MIS-OK-LIM*). As a rule, we will use the notation $\{x, y\}$ to refer to a tree created with a value of x for *GENERALIZE-NUMBER* and a value of y for *MIS-OK-LIM*.

The size of the hierarchies created by UNIMEM depended on the value of *GENERALIZE-NUMBER* more than on any other factor. When this value was low (e.g. 2), hierarchies were created that contained from four to six times as many generalizations as there were original training examples. When the value was high (e.g. 8), hierarchies contained many fewer generalizations. Figures B.1 and B.2 show growth rates for two UNIMEM trees trained under identical circumstances except for the value of *GENERALIZE-NUMBER*. The horizontal axis in these figures shows instances and the vertical axis shows generalizations. As illustrated by these figures, the rate of growth for UNIMEM trees ranged from exponential (figure B.2) to wildly varying nonmonotonic (figure B.1). The nonmonotonic growth is due to UNIMEM's practice of detecting and deleting erroneous generalizations [Lebowitz, 1982].

Table B.1 compares RECODER to UNIMEM on a variety of training sets and test sets. Each entry in the leftmost column of the table names a testing configuration consisting of a particular set of test words (either FAM or UNFAM) and a particular set of training words (from sets 1 through 6) The next column over shows RECODER's performance in that configuration according to equation B.1 to a given precision; that is, what percentage of test instances retrieved cases for which equation B.1 yielded a score higher than a given value. The next column shows the best UNIMEM performance for that configuration along with parameter settings for *GENERALIZE-NUMBER* and *MIS-OK-LIM* and an indication of

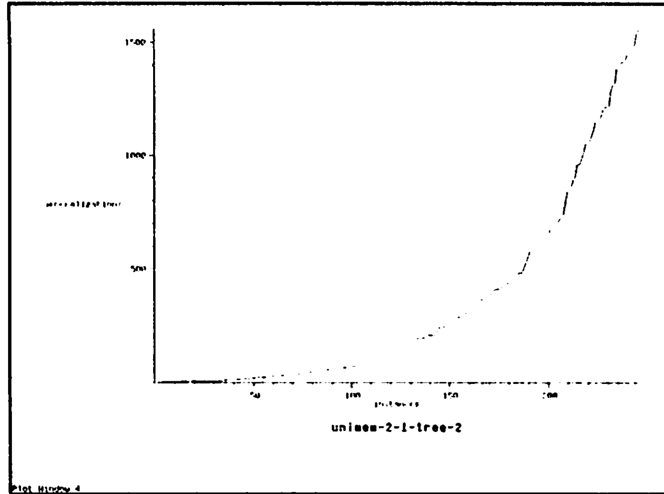

Figure B.2: History for growth of UNIMEM tree {2, 1}, trained on set 2.

whether the test sequence was reversed. The last column shows similar information for the worst UNIMEM performance for that configuration. Similarity of retrieved cases is compared to a precision of 0.35.

To get another perspective on the difference between the performance of RECODER and UNIMEM, figure B.3 compares RECODER's performance when trained on set 2 and tested on UNFAM to UNIMEM's best and worst performances on the same training/test configuration. The horizontal axis in this figure shows a sequence of test instances, and the vertical axis shows relative performance for each test instance. The performance of RECODER is shown in the figure by an unbroken dark line, that of UNIMEM at its best by a dashed line and that of UNIMEM at its worst by an unbroken gray line. This figure illustrates graphically what table B.1 depicts in numbers: that RECODER's overall performance is at least as good as UNIMEM's in all cases, and is noticeably better than UNIMEM at its worst.

In general, the better UNIMEM performed in retrieving cases, the larger the memory it had constructed. This is illustrated in figure B.3, which shows comparisons between RECODER and UNIMEM for the trees whose growth rates are shown in figures B.1 and B.2. These illustrate a general trend: the smaller the setting of *GENERALIZE-NUMBER*, the better the performance of UNIMEM. From figures B.1 and B.2, we see that the smaller the parameter settings, the larger is the resulting memory.

Why does UNIMEM not perform as well as RECODER? Often, it is because the generalizations that UNIMEM creates do not match well with the test instances presented for retrieval. The following example will illustrate an extreme example of this problem, using tree $\{8, 1\}$ trained on set 2, and tested on UNFAM. The growth

Table B.1: Comparing UNIMEM's best and worst performance to RECODER.

Configuration	*RECODER*	*UNIMEM best*	*UNIMEM worst*
FAM,1	100.0	2,1 94.949	8,1 27.659
FAM,2	78.787	2,1 69.696	8,1 7.608
FAM,3	75.757	2,1 63.636	8,1 20.833
FAM,4	79.797	2,1 72.727	8,1 43.434
FAM,5	80.808	2,1 65.656	5,1 53.608
FAM,6	75.757	2,1 57.142	8,1 42.424
UNFAM,1	77.777	2,1 56.565	8,1 6.451
UNFAM,2	74.7474	2,1 56.565	8,1 2.061
UNFAM,3	77.7777	2,1 58.585	8,1 16.483
UNFAM,4	83.8383	2,1 58.585	8,1 41.414
UNFAM,5	85.8585	2,1 70.707	5,1 59.183
UNFAM,6	81.8181	2,1 69.387	8,1 53.535

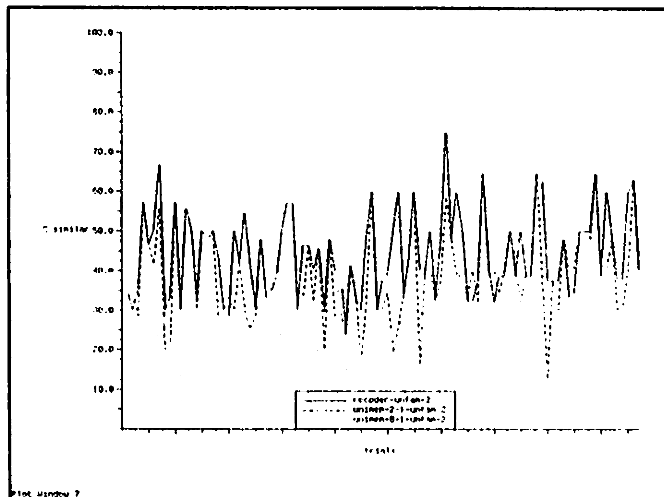

Figure B.3: Comparing RECODER with UNIMEM best and worst.

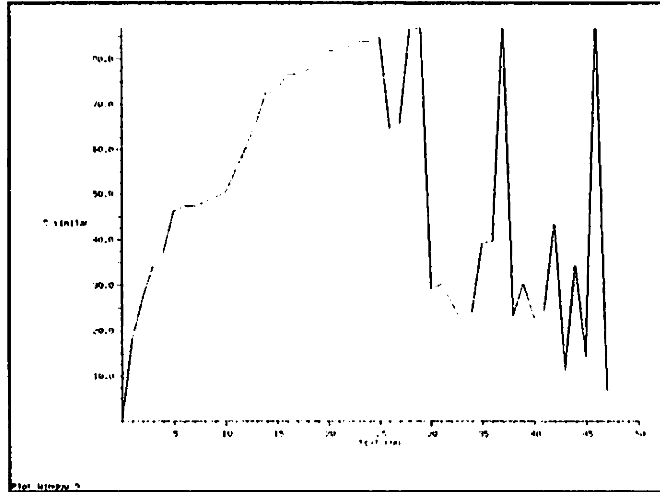

Figure B.4: "Goodness" history for a generalization tree.

history of this tree is shown in figure B.1, and its results are shown in figure B.3. Figure B.4 shows the growth rate of the tree for this example, along with a history of the "goodness" of the cases retrieved. The "goodness" history was obtained by running the testing set on the tree after every fifth training instance had been assimilated, and recording the percentage of retrievals whose similarity (as measured by equation B.1) was better than 0.3. Figures B.5 and B.6 show histories of the actual similarity results for two test words, again sampled after every fifth training instance. The most remarkable aspect of these histories is the almost uniform lack of good results when generalizations are present. We will look closely at segments of the histories of two words, "tail" and "vault" (chosen because they illustrate a range of degenerate effects), to see why this happens.

To start, we will look at a segment of the UNIMEM tree after 200 training instances have been examined. At this point, there are 9 generalizations in the tree, including the three shown below. Each generalization is shown by its name, its parent, the features used as indices, and the words indexed by that generalization (the root node has "*" listed as parent and shows the number of words indexed, rather than the words themselves). Each feature shows the feature's name followed by its value, the number of sibling generalizations that share that feature, and a confidence measure for that feature (these are described more fully in [Lebowitz, 1986]). For instance, the generalization NODE37 has as its parent NODE31, indexes the words "trifle" and "double," and uses the features 1-AFTER-L and 0-AFTER-L as indices. 1-AFTER-L has a value of E, has 1 sibling with that feature (itself), and has a confidence measure of 0.

node: NODE2

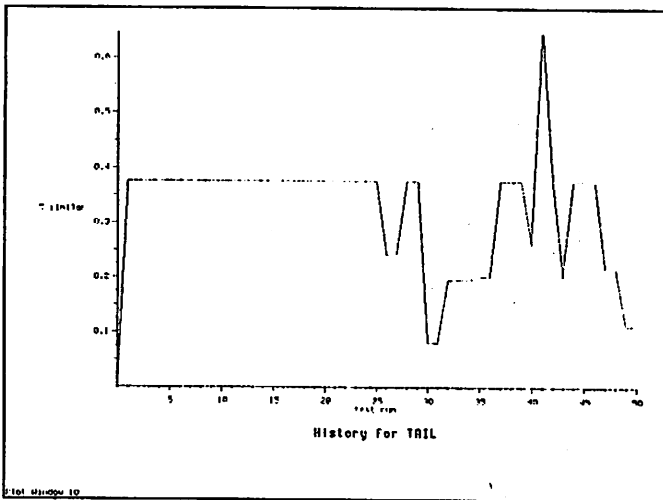

Figure B.5: Similarity history for "tail."

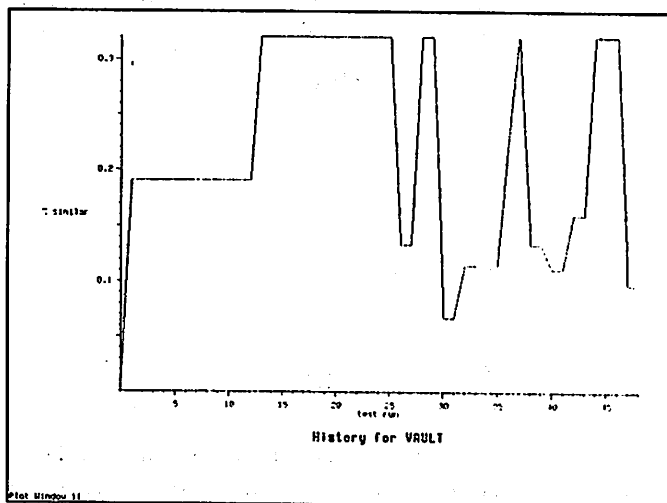

Figure B.6: Similarity history for "vault."

parent: *
[*(155 words)*]

node: NODE31
parent: NODE2

1-AFTER-C	E	[2]	(-2)
0-AFTER-E	E	[2]	(7)
0-AFTER-C	C	[2]	(4)
0-AFTER-A	A	[1]	(4)
0-AFTER-R	R	[2]	(6)
0-AFTER-T	T	[2]	(4)

[tracer]

node: NODE37
parent: NODE31

1-AFTER-L	E	[1]	(0)
0-AFTER-L	L	[1]	(0)

[trifle double]

The words retrieved from this tree are not very similar to the test words. Both words match NODE31 and NODE37,[2] and for both words, the most similar case indexed by NODE37 (the most specific generalization that matches) is "trifle." For "tail," equation B.1 yields a degree of match of 0.258; for "vault," the degree of match is 0.111.

After five more training instances have been assimilated, the resulting tree has a total of 9 generalizations. NODE31 and NODE37 remain, and a new generalization, NODE38, has been added, as shown below.

node: NODE2
parent: *
[*(155 words)*]

node: NODE31
parent: NODE2

1-AFTER-C	E	[1]	(-2)

[2] An instance matches a generalization if there is at least one common feature between them, and if for all common features, the number of contradictions (i.e. features with non-matching values) is no greater than *MIS-OK-LIM*. In this example, *MIS-OK-LIM* has a value of 1. Thus, in NODE31, for both "tail" and "vault," the features 1-AFTER-C, 0-AFTER-E, 0-AFTER-C, and 0-AFTER-A are ignored; features 0-AFTER-A and 0-AFTER-T have values that match. For NODE37, both "tail" and "vault" match feature 0-AFTER-L; "tail" doesn't have a feature 1-AFTER-L; "vault" does have that feature, and its value of T contradicts the generalization's value of E, but that leaves the total number of contradictions at 1, which is not greater than *MIS-OK-LIM*.

0-AFTER-E	E	[1]	(11)
0-AFTER-C	C	[1]	(4)
0-AFTER-A	A	[1]	(6)
0-AFTER-R	R	[1]	(8)
0-AFTER-T	T	[1]	(8)

[divide tracer]

node: NODE37
parent: NODE31

1-AFTER-L	E	[1]	(0)
0-AFTER-L	L	[1]	(1)

[double]

node: NODE38
parent: NODE37

2-AFTER-T	I	[1]	(0)
0-AFTER-I	I	[1]	(0)

[retail trifle]

When "tail" is tested on the resulting tree, NODE38 is returned as a valid generalization, since it contains features 2-AFTER-T and 0-AFTER-I, and both values match. "Retail" is selected as the most relevant word, with a similarity measure of 0.645 (this is a good similarity measure). When "vault" is tested, NODE38 does not match since there are no features in common. Now, since only the word "double" is indexed by NODE37, it is returned as the best match; as it turns out, the similarity measure for "vault" and "double" is 0.111.

Five more training instances yield a tree with only two generalizations: NODE2 (which is the root of the tree and is never deleted) and NODE42. The tree is shown below.

node: NODE2
parent: *
[*(156 words)*]

node: NODE42
parent: NODE2

3-AFTER-T	N	[1]	(-2)
2-AFTER-H	N	[1]	(0)
1-AFTER-A	N	[1]	(0)
0-AFTER-N	N	[1]	(1)
2-AFTER-T	A	[1]	(-2)
1-AFTER-H	A	[1]	(0)

```
0-AFTER-A            A    [1]    (0)
1-AFTER-T            H    [1]    (-2)
0-AFTER-H            H    [1]    (0)
0-AFTER-T            T    [1]    (1)
[methane than]
```

By default, all instances match the root node; since "tail" does not match NODE42 (the features 3-AFTER-T, 1-AFTER-A, 2-AFTER-T, and 1-AFTER-T all contradict), it returns the best-matching word from the root node, which happens to be "aid" with a similarity of 0.375. "Vault" does match NODE42, since the only shared features are 1-AFTER-A, 0-AFTER-A, and 0-AFTER-T, and the only contradiction is 1-AFTER-A. The best matching word in NODE42 is "than," with a similarity of 0.16.

After a total of 230 training instances have been encountered, it turns out that all generalizations (except for the root node) have been deleted. From the 158 words indexed by the root node, "aid" is chosen as the best match for "tail" with a similarity of 0.375, and "malt" is chosen as the best match for "vault," with a similarity of 0.32. As it turns out, the match for "vault" is better than any match encountered up to this point in the example. The match for "tail" is surpassed only by "retail."

After five more training instances have been encountered, a tree with 4 generalizations results. One of the new generalizations, NODE61, is shown below.

```
node: NODE2
parent: *
[(156 words)]

    node: NODE61
    parent: NODE2
    1-AFTER-S            T    [1]    (-1)
    0-AFTER-T            T    [2]    (1)
    0-AFTER-S            S    [1]    (1)
    2-AFTER-A            I    [2]    (-1)
    2-AFTER-I            N    [2]    (-2)
    0-AFTER-N            N    [3]    (1)
    0-AFTER-A            A    [2]    (1)
    0-AFTER-I            I    [3]    (2)
    [theater obligee station pianist]
```

Both "tail" and "vault" match NODE61. Of the four words indexed there, "tail" best matches "station" with a similarity of 0.212; "vault" best matches "theater" with a similarity of 0.097. Both of these similarity measures are worse than those for words obtained from the root.

When five more training words have been added, the resulting tree contains a total of 18 generalizations. In particular, several subgeneralizations have been added under NODE61, some of which are shown below.

node: NODE61
parent: NODE2

0-AFTER-T	T	[2]	(2)
0-AFTER-S	S	[1]	(5)
2-AFTER-I	N	[2]	(-2)
0-AFTER-N	N	[3]	(1)
0-AFTER-A	A	[2]	(2)
0-AFTER-I	I	[3]	(2)

[station]

node: NODE65
parent: NODE61

0-AFTER-R	R	[1]	(2)
0-AFTER-E	E	[2]	(2)

[obscure]

node: NODE68
parent: NODE65

4-AFTER-P	E	[1]	(0)
0-AFTER-P	P	[1]	(0)

[pervert process]

node: NODE67
parent: NODE65

4-AFTER-E	R	[1]	(0)
1-AFTER-E	R	[1]	(0)
3-AFTER-E	E	[1]	(0)
2-AFTER-E	T	[1]	(0)

[pervert theater]

As can be seen, neither "tail" or "vault" match NODE67, since no features are shared; in particular, this means that "vault" has no chance of retrieving "theater," its previous best match from NODE61. Instead, it returns "station," which with a similarity of 0.095 is slightly worse than the similarity to "theater." "Tail" again retrieves "station," with a similarity of 0.216.

Finally, we show a segment of the tree after 245 training instances have been encountered. The complete tree has 22 generalizations, but of primary concern to us is NODE84, which has been added under NODE61.

node: NODE2
parent: *
[*(157 words)*]

node: NODE62
parent: NODE2

5-AFTER-I	E	[1]	(-1)
4-AFTER-N	E	[1]	(0)
0-AFTER-E	E	[1]	(6)
2-AFTER-N	O	[1]	(0)
0-AFTER-O	O	[1]	(7)
0-AFTER-N	N	[2]	(2)
0-AFTER-I	I	[2]	(4)

[traitor invoke]

node: NODE72
parent: NODE62

1-AFTER-I	N	[3]	(-1)
3-AFTER-I	O	[3]	(-1)
2-AFTER-O	E	[3]	(-1)
4-AFTER-E	R	[1]	(0)
1-AFTER-E	R	[1]	(0)
0-AFTER-R	R	[2]	(3)
3-AFTER-E	E	[1]	(0)
2-AFTER-E	T	[1]	(0)
0-AFTER-T	T	[1]	(2)

[obscure theater]

node: NODE61
parent: NODE2

0-AFTER-T	T	[1]	(6)
0-AFTER-S	S	[1]	(7)
0-AFTER-N	N	[2]	(3)
0-AFTER-A	A	[1]	(6)
0-AFTER-I	I	[2]	(5)

[]

node: NODE84
parent: NODE61

2-AFTER-A	I	[1]	(0)
1-AFTER-T	I	[1]	(0)
1-AFTER-A	T	[1]	(0)

[satisfy station]

NODE84 has a dual effect: it removes "station" as a word indexed by NODE61 (which both "tail" and "vault" match), and it adds the word to its own index set. This has the net effect of causing "station" to be inaccessible to both "tail" and "vault," since neither word matches NODE84. For "tail," this results in a word from another matching generalization (NODE62) to be returned. The word returned is "theater," which at a similarity of 0.111 is the worst match seen so far for "tail." "Vault," however, has a worse fate: since NODE61 is the only generalization that matches it, and since NODE61 indexes no words, no words are returned as a match for "vault."

The above is a rather extreme example, and it could be argued that there were simply never enough generalizations in the tree to give adequate coverage to the test instances. To a certain degree, this is true. In fact, several factors conspire here to make generating a good tree of a reasonable size quite difficult. First, the number of features varies widely from word to word. In order to capture regularities that appear in words with a small number of features, *GENERALIZE-NUMBER* must be set so low that tree growth becomes exponential. If one wants reasonable growth, *GENERALIZE-NUMBER* must be set higher, excluding regularities in words with few features (this second path leads to extreme examples of the kind shown above).

Secondly, the order of presentation might contribute to poor performance. As the above example shows, potentially useful generalizations may get deleted during the course of training. Different orderings of the training set could result in more accurate retrievals.

Finally, the set of features itself could be contributing to UNIMEM's poor generalizations. In the bias used above, the same letter appearing more than once in the same word will result in a set of contradictory features, making generalization difficult. For example, the first letter after the first "t" in "letter" is a "t," while the first letter after the second "t" in "letter" is an "e;" this results in two contradictory values ("t" and "e") for the feature 1-AFTER-T.

These factors will be addressed in the next section.

B.3 Phase Two

In the second phase of the experiment, only words four and five letters long were used in training mode. One large set of words, called set 45, was created from all of the four and five letter words in the six original sets. Set 45 contained a total of 658 words. The FAM test set was similarly modified to contain only four and five letter words, yielding a total of 34 words (the UNFAM set was not modified, since all of its 99 words were already four or five letters long).

In addition, UNIMEM was trained on three different presentations of set 45.

The second presentation was the reverse of the original presentation, and the third presentation was the same as the original, but with the first 102 words moved to the middle of the group (specifically, following the 357th word of the original group).

In the first phase of the experiment, only one feature bias was considered. The second phase considered two others as well. The first of these new biases, called the "position-only" bias, represents a word with one feature $< n >$ per letter; n is the absolute position of the letter in the word. The second new bias, called the "letters-after" bias, represents a word as a set of features $< x >$: each x is a letter in the word whose value is a letter appearing one or more positions after x. To compare representations with each of these biases, we first repeat the representation for "cat" in the original bias.

{(<C 0> C) (<C 1> A) (<C 2> T) (<A 0> A) (<A 1> T) (<T 0> T)}

Next, "cat" is represented in the "position-only" bias:

{(<1> C) (<2> A) (<3> T)}

Finally, "cat" is represented in the "letters-after" bias:

{(<C> A) (<C> T) (<A> T)}

The second phase of the experiment also compares the efficiency of retrieval using UNIMEM to that of retrieval using RECODER. For UNIMEM, we use as a measure of efficiency the sum of the total number of generalizations examined and the cases indexed by returned generalizations. We include cases in our measure because UNIMEM must examine all cases in a returned generalization to determine which is the best match. We measure RECODER's efficiency by assuming a parallel retrieval algorithm which, as explained in chapter 5, has a performance logarithmic in the total number of cases stored.The arrangement of arbitration nodes described in chapter 5 will yield a performance of $2\lceil \log_2(n) \rceil$, where n is the number of stored cases; this is the figure we use for RECODER's performance estimates.

Table B.2 shows UNIMEM's performance when trained on set 45 using the original feature bias. The columns labeled "accuracy" show the percentage of retrieved cases that were similar to a precision of 0.35. The columns labeled "efficiency" show the number of nodes examined during retrieval, averaged out over all words in the test sets. By contrast, RECODER performed at an accuracy of 100.0 for FAM and 96.97 for UNFAM, and had an efficiency of $2\lceil \log_2(658) \rceil = 20$ per retrieval, at least as good across test sets as any version of UNIMEM.

Table B.2 shows fairly clearly the relation between memory size and retrieval accuracy: those trees with more than twice as many generalizations as instances performed consistently better than trees with fewer generalizations than instances. Interestingly, while an unrealistically large memory seemed to ensure good performance, a small memory did not necessarily ensure poor performance. Tree $\{8, 0\}$ had only two generalizations, and it had the best overall performance apart from the four largest trees. Conversely, the tree with the fifth largest memory (tree $\{7, 1\}$)

Table B.2: Comparing retrieval with original bias, at a precision of 0.35.

		FAM		*UNFAM*	
Tree	*Nodes*	*Accuracy*	*Efficiency*	*Accuracy*	*Efficiency*
6,0	3263	94.118	84.588	83.838	67.333
6,1	2838	97.059	76.794	85.858	57.556
7,0,R	2003	100.0	58.088	85.858	69.899
7,1,R	1432	100.0	49.029	83.838	55.828
7,1	404	67.647	27.706	65.656	33.394
8,1,M	342	70.588	57.824	66.667	50.687
8,1	134	76.470	159.882	67.677	110.636
8,0,R	90	91.176	18.912	35.353	31.303
8,1,R	47	82.353	34.236	38.776	57.694
8,0	2	88.235	284.765	75.758	273.364

performed fairly poorly overall. And tree $\{8,0,R\}$,[3] with 90 generalizations, performed fairly well on the FAM set of words and very poorly on the UNFAM set.

Each of these trees is worth examining in more detail. Tree $\{8,0,R\}$ performed as well as it did on the FAM set because that set contained only words on which the tree was trained, and the tree retained most of the generalizations involving those words. The UNFAM set, on the other hand, contained *no* words used in training; the dissimilarities between the UNFAM words and the words contained in retrieved generalizations resulted in poor performance.

In tree $\{7,1\}$, performance was poor on both training sets for the same reason that performance was poor on UNFAM for tree $\{8,0,R\}$: the words contained in retrieved generalizations were not very similar to the training words. This may seem surprising in the case of FAM, since all words in that set were used in training tree $\{7,1\}$. However, of the 34 words from FAM, 9 did not appear in the final tree at all, having been involved only in generalizations that were eventually discarded; the other 25 appeared only in the root of the tree. Since UNIMEM retrieves cases by returning the deepest matching generalizations, the root will only be retrieved if no other generalizations match. Since there were always generalizations that matched when testing tree $\{7,1\}$ on FAM, these were always returned, resulting in poor overall performance.

Interestingly, tree $\{7,1\}$ would have performed significantly better on FAM if it had only contained the root (which indexed 180 words). To verify this, a run was made on tree $\{7,1\}$ with all generalizations deleted except for the root —

[3]The R denotes that the presentation order of the training set was reversed in creating this tree.

performance on FAM was 94.118% at a precision of .35. This is as good as the performance on FAM of tree {6,0}, which contained 3263 generalizations.

The above situation explains the relatively good performance of tree $\{8, 0, R\}$. This tree contained only 2 generalizations, one of which was the root. The root, however, indexed 332 words, and was retrieved 29 out of 34 times when testing FAM, and 81 out of 99 times when testing UNFAM. Of those, all 29 words retrieved for FAM were similar to the originals with a precision of .35, while 73 of the words retrieved for UNFAM were similar to the originals with the same precision. Since our version of UNIMEM picks the best match from the cases returned to it by applying the similarity metric to each case and returning the best match, UNIMEM's results here are as good as they are because UNIMEM is essentially duplicating RECODER's algorithm.[4] That is, UNIMEM achieved its high performance here by examining many cases and choosing the best match, rather than by taking advantage of a generalization hierarchy. As seen above, a generalization hierarchy could actually *impede* UNIMEM's overall performance.

Tables B.3 and B.4 show UNIMEM's performance when trained on set 45 using the "position-only" and "letters-after" biases, respectively. RECODER has 100.0 retrieval accuracy at the given precisions for each of the FAM sets for both biases. RECODER's retrieval accuracy for the UNFAM set was 85.858 for the "position-only" bias and 94.949 for the "letters-after" bias. Efficiency for RECODER was again $2\lceil \log_2(658) \rceil = 20$. As with the original bias, RECODER's accuracy was at least as good as UNIMEM's in every instance. RECODER's efficiency was similar to most configurations of UNIMEM for the "position-only" bias, and was at least as good across training sets as the majority of configurations for the "letters-after" bias.

The trees created by UNIMEM using the "position-only" bias were more stable than trees created using the other biases. This stability is reflected in the size of the trees, their relative retrieval accuracy, and their relative efficiency. All of these trees contain fewer generalizations than there were cases in the training set, and the largest of the trees is contains only four times as many nodes as the smallest. This is in marked contrast to trees created with the original bias, where the largest tree contained five times as many nodes as there were training instances and over 1500 times as many nodes as were contained in the smallest tree. Eight out of the ten trees created with the "position-only" bias were nearly as efficient as UNIMEM in retrieving cases; eight out of the ten trees were also able to retrieve cases for words in the FAM test set that were nearly as similar to those words as the cases retrieved by RECODER.

Unfortunately, this consistency has a negative manifestation as well. Though the various trees retrieved cases that were consistently similar to words in the UN-

[4]This is our own extension to UNIMEM that enables it to return a single case. The original version of UNIMEM would have returned a set of generalizations, each indexing a set of cases.

Table B.3: Comparing retrieval with positional bias, at a precision of 0.5.

		FAM		*UNFAM*	
Tree	*Nodes*	*Accuracy*	*Efficiency*	*Accuracy*	*Efficiency*
1,1	282	94.118	21.941	74.747	25.273
1,1,R	275	97.059	21.147	67.677	23.656
2,0	495	97.059	27.647	69.697	32.899
2,0,R	439	100.0	24.500	65.656	28.778
2,1	254	94.118	23.706	69.697	20.697
2,1,R	278	100.0	24.529	71.717	25.677
3,1	237	79.412	22.559	65.656	25.030
3,1,R	222	100.0	22.235	61.616	33.030
4,1	125	70.588	65.765	58.586	45.465
4,1,R	190	97.059	65.000	58.586	52.020

FAM test set, these cases were all noticeably less similar than cases retrieved by RECODER (where the retrieval accuracy was 85.858). This points up a significant weakness manifested in all of the experiments involving UNIMEM: the trees that were created were uniformly poor at retrieving cases for unfamiliar situations.

The trees created using the "letters-after" bias point out the importance of choosing a good representation for clustering algorithms such as UNIMEM. If little thought is given to representations, situations arise where small perturbations to the clustering algorithm or the form in which it receives its input cause large perturbations in performance. Such is the case here. Simply reversing the order of presentation of the training data causes large fluctuations in retrieval accuracy (consider, for example, trees $\{3,2\}$ and $\{3,2,R\}$). Small changes in UNIMEM's *GENERALIZE-NUMBER* parameter can likewise cause large changes in both performance and efficiency, as can be seen from trees $\{3,1\}$ and $\{4,1\}$. As with all biases, good performance on familiar data does not necessarily translate to good performance on unfamiliar data, as witnessed in tree $\{3,10,R\}$. Note also that even though several trees were able to perform at least as efficiently as RECODER across test sets (trees $\{2,1\}$, $\{3,10\}$, $\{3,2,R\}$, $\{3,10,R\}$, and perhaps $\{3,1\}$), none of these trees gave accurate performance across test sets. Most of these performed fairly poorly for both test sets; the exceptions, trees $\{3,2,R\}$ and $\{3,10,R\}$, performed well on the FAM set but poorly on the UNFAM set (for which RECODER had a retrieval accuracy of 94.949).

In summary, the use of algorithms that construct generalization hierarchies calls for many decisions to be made and many possibilities to be explored. The choice of representation for features is crucial to the performance of the resulting tree; a

Table B.4: Comparing retrieval with "letters-after" bias, at a precision of 0.35.

		FAM		*UNFAM*	
Tree	*Nodes*	*Accuracy*	*Efficiency*	*Accuracy*	*Efficiency*
2,1	382	35.294	10.824	27.273	13.121
3,1	347	47.059	19.147	42.424	26.960
3,2	338	41.176	22.470	48.485	34.1919
3,10	335	58.823	21.000	36.363	19.232
4,1	44	82.353	215.089	60.606	169.677
4,2	45	61.765	141.353	63.636	173.889
4,5	45	52.941	96.059	54.545	101.828
4,10	46	58.824	108.059	52.525	122.687
3,1,R	308	70.589	26.059	49.495	24.596
3,2,R	329	91.176	15.882	60.606	15.263
3,10,R	308	94.118	16.353	51.515	19.364
4,1,R	75	93.939	158.273	80.412	217.887

bad representation can cause different orderings of training data or small changes in parameter settings to yield large fluctuations in the performance of the resulting trees. Even relatively good representations offer no significant advantages over a retrieval scheme based on massive parallel search. Most significantly, the training data itself creates biases that hinder good retrievals for unfamiliar test data. Stated briefly, the difficulty is in knowing in advance which of the many generalizations that can be made will be useful in the future. The approach taken by RECODER is to avoid making explicit generalizations at all, and to get the effects of generalization by using a partial matching algorithm when retrieving cases.

Bibliography

[Alterman, 1988] R. Alterman. Adaptive planning. *Cognitive Science*, 12:393–421, 1988.

[Anderson, 1986] J.R. Anderson. Knowledge compilation: The general learning mechanism. In R. S. Michalski, J. G. Corbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 2, pages 289–310. Morgan Kaufmann, Los Altos, CA, 1986.

[Ashley and Rissland, 1986] K. D. Ashley and E. L. Rissland. Compare and contrast, a test of expertise. In *Proceedings of the Sixth National Conference on Artificial Intelligence*. American Association for Artificial Intelligence, Morgan Kaufmann Publishers, Inc., 1986. 95 First Street, Los Altos, CA 94022.

[Ashley, 1988] K. D. Ashley. *Modelling Legal Argument: Reasoning with Cases and Hypotheticals*. PhD thesis, University of Massachusetts, 1988. Department of Computer and Information Science.

[Barletta and Mark, 1988] M. Barletta and W. Mark. Explanation-based indexing of cases. In *Proceedings of the 1988 Case-Based Reasoning Workshop*, Clearwater Beach, Florida, 1988.

[Barstow, 1979] D. R. Barstow. An experiment in knowledge-based automatic programming. *Artificial Intelligence*, 12:73–119, 1979.

[Bonar, 1985] J. G. Bonar. *Understanding the Bugs of Novice Programmers*. PhD thesis, University of Massachusetts, 1985. Department of Computer and Information Science.

[Bradtke and Lehnert, 1988] S. Bradtke and W. Lehnert. Some experiments with case-based search. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 133–138. American Association for Artificial Intelligence, Morgan Kaufmann Publishers, Inc., 1988. 95 First Street, Los Altos, CA 94022.

[Burstein, 1986] M.H. Burstein. Concept formation by incremental analogical reasoning and debugging. In R. S. Michalski, J. G. Corbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 2, pages 351–369. Morgan Kaufmann, Los Altos, CA, 1986.

[Carbonell, 1983] J. G. Carbonell. Learning by analogy: Formulating and generalizing plans from past experience. In R. S. Michalski, J. G. Corbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 1. Morgan Kaufmann, Los Altos, CA, 1983.

[Carbonell, 1986] J. G. Carbonell. Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In R. S. Michalski, J. G. Corbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 2. Morgan Kaufmann, Los Altos, CA, 1986.

[Carbonell, 1989] J. G. Carbonell. Paradigms for machine learning. *Artificial Intelligence*, 40:1–9, 1989.

[Charniak *et al.*, 1980] E. Charniak, C. K. Riesbeck, and D. V. McDermott. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1980.

[DeJong and Mooney, 1986] G. F. DeJong and R. Mooney. Explanation-based learning: an alternate view. *Machine Learning*, 1:145–176, 1986.

[Dershowitz, 1986] N. Dershowitz. Programming by analogy. In R. S. Michalski, J. G. Corbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 2. Morgan Kaufmann, Los Altos, CA, 1986.

[Feigenbaum, 1963] E. A. Feigenbaum. The simulation of verbal learning behaviour. In E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 297–309. McGraw-Hill, New York, NY, 1963.

[Fikes and Nilsson, 1971] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[Fikes *et al.*, 1972] R. E. Fikes, P. E. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[Fisher, 1987] D. H. Fisher. Knowledge acquisition via incremental concept clustering. *Machine Learning*, 2 (2):139–172, 1987.

[Friedman, 1974] D. Friedman. *The Little LISPer (First Edition)*. Science Research Associates, Inc., Chicago, 1974.

[Gentner and Toupin, 1986] D. Gentner and C. Toupin. Systematicity and surface similarity in the development of analogy. *Cognitive Science*, 10:277–300, 1986.

[Gentner, 1983] D. Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7:155–170, 1983.

[Hammond, 1986a] K. Hammond. *Case-Based Planning: An Integrated Theory of Planning, Learning, and Memory.* PhD thesis, Yale University, 1986. Department of Computer Science 488.

[Hammond, 1986b] K. J. Hammond. Chef: A model of case-based planning. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 267–271, 95 First Street, Los Altos, CA 94022, 1986. American Association for Artificial Intelligence, Morgan Kaufmann Publishers, Inc.

[Hammond, 1986c] K. J. Hammond. Learning to anticipate and avoid planning problems through the explanation of failures. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 556–560, 95 First Street, Los Altos, CA 94022, 1986. American Association for Artificial Intelligence, Morgan Kaufmann Publishers, Inc.

[Hammond, 1986d] K. J. Hammond. The use of remindings in planning. In *Proceedings of the Eighth Annual Cognitive Science Society Conference*, pages 442–451, Hillsdale, NJ, 1986. Cognitive Science Society, Lawrence Erlbaum Associates.

[Hammond, 1987] K. J. Hammond. Explaining and repairing plans that fail. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 109–114. American Association for Artificial Intelligence, Morgan Kaufmann Publishers, Inc., 1987.

[Hammond, 1988] K. J. Hammond. Case-based planning. In *Proceedings of the 1988 Case-Based Reasoning Workshop*, Clearwater Beach, Florida, 1988.

[Hammond, 1989] K. J. Hammond. *Case-Based Planning: Viewing Planning as a Memory Task.* Academic Press, Inc., San Diego, CA, 1989.

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

[Holyoak and Thagard, 1989] K.J. Holyoak and P. Thagard. Analogical mapping by constraint stisfaction. *Cognitive Science*, 13:295–355, 1989.

[Kolodner *et al.*, 1985] J. L. Kolodner, Jr. R. L. Simpson, and K. Sycara-Cyranski. A process model of case-based reasoning in problem solving. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, 95 First Street, Los Altos, CA 94022, 1985. American Association for Artificial Intelligence, Morgan Kaufmann Publishers, Inc.

[Kolodner, 1984] J. L. Kolodner. *Retrieval and Organizational Strategies in Conceptual Memory: A Computer Model.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1984.

[Kolodner, 1988] J. L. Kolodner. Retrieving events from a case memory: A parallel implementation. In *Proceedings of the 1988 Case-Based Reasoning Workshop*, pages 233–249, Clearwater Beach, FL, 1988. Defense Advanced Research Projects Agency, Morgan Kaufman.

[Kolodner, 1989] J. L. Kolodner. Selecting the best case for a case-based reasoner. In *Proceedings of the Eleventh Annual Cognitive Science Society Conference*, pages 155–162, Hillsdale, NJ, 1989. Cognitive Science Society, Lawrence Erlbaum Associates.

[Korf, 1985] R. E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26:35–77, 1985.

[Laird *et al.*, 1986] J.E. Laird, P.S. Rosenbloom, and A. Newell. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986.

[Laird *et al.*, 1987] J.E. Laird, A. Newell, and P.S. Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[Lebowitz, 1982] M. Lebowitz. Correcting erroneous generalizations. *Cognition and Brain Theory*, 5(4):367–381, 1982.

[Lebowitz, 1983] M. Lebowitz. Memory-based parsing. *Artificial Intelligence*, 21:363–404, 1983.

[Lebowitz, 1986] M. Lebowitz. Concept learning in a rich input domain: Generalization-based memory. In R. S. Michalski, J. G. Corbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 2. Morgan Kaufmann, Los Altos, CA, 1986.

[Lebowitz, 1988] M. Lebowitz. Deferred commitment in unimem: Waiting to learn. In *Proceedings of the Fifth International Conference on Machine Learning*. Morgan Kaufman, 1988.

[Lehnert, 1987a] W. G. Lehnert. Case-based problem solving with a large knowledge base of learned cases. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 301–306. American Association for Artificial Intelligence, Morgan Kaufmann Publishers, Inc., 1987. 95 First Street, Los Altos, CA 94022.

[Lehnert, 1987b] W. G. Lehnert. Learning to integrate syntax and semantics. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 179–190. Morgan Kaufman, 1987.

[Luria, 1988] M. Luria. *Knowledge Intensive Planning.* PhD thesis, University of California Berkeley, 1988. Report No. UCB/CSD 88/433.

[Manna and Waldinger, 1981] Z. Manna and R. Waldinger. A deductive approach to program synthesis. In B. L. Webber and N. J. Nilsson, editors, *Readings in Artificial Intelligence*, pages 141–172. Tioga, Palo Alto, CA, 1981.

[Michalski, 1986] R. S. Michalski. Understanding the nature of learning: Issues and research directions. In R. S. Michalski, J. G. Corbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 2. Morgan Kaufmann, Los Altos, CA, 1986.

[Mitchell *et al.*, 1983] T. M. Mitchell, P. E. Utgoff, and R. B. Banerji. Learning by experimentation: Acquiring and refining problem-solving heuristics. In R. S. Michalski, J. G. Corbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 1. Morgan Kaufmann, Los Altos, CA, 1983.

[Mitchell *et al.*, 1986] T. M. Mitchell, R.M. Keller, and S.T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1:47–80, 1986.

[Plan, 1989] Darpa Machine Learning Program Plan. Case-based reasoning. In *Proceedings of the 1989 Case-Based Reasoning Workshop*, Pensacola Beach, Florida, 1989.

[Quillian, 1968] M. R. Quillian. Semantic memory. In M. L. Minsky, editor, *Semantic Information Processing*, pages 227–270. MIT Press, Cambridge, MA, 1968.

[Quinlan, 1983] J. R. Quinlan. Learning efficient classification procedures and their application to chess end games. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 463–482. Tioga, Palo Alto, CA, 1983.

[Rich, 1981] C. Rich. A formal representation for plans in the programmer's apprentice. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence.* American Association for Artificial Intelligence, Morgan Kaufmann Publishers, Inc., 1981. 95 First Street, Los Altos, CA 94022.

[Riesbeck and Martin, 1985] C. Riesbeck and C. Martin. Direct memory access parsing. Technical Report Reasearch Report #354, Yale University, 1985.

[Riesbeck and Schank, 1989] C. K. Riesbeck and R. C. Schank. *Inside Case-Based Reasoning.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

[Rissland and Ashley, 1987] E. L. Rissland and K. D. Ashley. Hypo: a case-based reasoning system. Technical Report CPTM-18, Department of Computer and Information Science, University of Massachusetts at Amherst, Amherst, MA, 01003, 1987.

[Rist, 1989] R. S. Rist. Schema creation in programming. *Cognitive Science*, 13:389–414, 1989.

[Rumelhart *et al.*, 1986] D.E. Rumelhart, G.E. Hinton, and J. L. McClelland. A general framework for parallel distributed processing. In D.E. Rumelhart, J.L. McClelland, and the PDP research group, editors, *Parallel Distributed Processing: Explorations in the Microstructure*, volume 1, pages 45–76. Bradford Books, Cambridge, MA, 1986.

[Schank, 1982] R. Schank. *Dynamic Memory: A Theory of Reminding and Learning in Computers and People.* Cambridge University Press, New York, NY, 1982.

[Shapiro, 1982] E. H. Shapiro. *Algorithmic Program Debugging.* PhD thesis, Yale University, 1982.

[Shinn, 1988] H. S. Shinn. Abstractional analogy: A model of analogical reasoning. In *Proceedings of the 1988 Case-Based Reasoning Workshop*, pages 370–387, Clearwater Beach, FL, 1988. Defense Advanced Research Projects Agency, Morgan Kaufman.

[Spohrer *et al.*, 1985] J. C. Spohrer, E. Soloway, and E. Pope. A goal/plan analysis of buggy pascal programs. Technical Report Research Report #392, Yale University, 1985.

[Stanfill and Waltz, 1986] C. Stanfill and D. Waltz. Toward memory-based reasoning. *Communications of the ACM*, 29, no. 12:1213–1228, 1986.

[Stanfill and Waltz, 1988] C. Stanfill and D. Waltz. The memory-based reasoning paradigm. In *Proceedings of the 1988 Case-Based Reasoning Workshop*, Clearwater Beach, Florida, 1988.

[Sycara, 1987] E. Sycara. *Resolving Adversarial Conflicts: An Approach Integrating Case-Based and Analytic Methods.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1987. Atlanta, GA.

[Thagard *et al.*, 1988] P. Thagard, K. Holyoak, G. Nelson, and D. Gochfeld. Analog retrieval by constraint satisfaction. 1988. Unpublished Manuscript.

[Waltz and Pollack, 1985] D. L. Waltz and J. B. Pollack. Massively parallel parsing: A strongly interactive model of natural language interpretation. *Cognitive Science*, 9:51–74, 1985.

[Waltz, 1989] D. L. Waltz. Is indexing used for retrieval? In *Proceedings of the 1989 Case-Based Reasoning Workshop*, Pensacola Beach, Florida, 1989.

[Waters, 1985] R. C. Waters. The programmer's apprentice: A session with KBEmacs. *IEEE Transactions on Software Engineering*, SE-11(11):1296–1320, 1985.

[Wilensky, 1987] R. Wilensky. Some problems and proposals for knowledge representation. Technical Report Report No. UCB/CSD 87/351, University of California Berkely, Berkeley, CA, 1987.

[Williams, 1988a] R. S. Williams. Learning to program by examining and modifying cases. In *Proceedings of the 1988 Case-Based Reasoning Workshop*, Clearwater Beach, Florida, 1988.

[Williams, 1988b] R. S. Williams. Learning to program by examining and modifying cases. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 318–324. Morgan Kaufman, 1988.