INTROSPECTIVE LEARNING FOR CASE-BASED PLANNING

by

Susan Fox

Submitted to the faculty of the Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the Department of Computer Science,

Indiana University

October, 1995

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

David B. Leake, Ph.D.
(Principal Advisor)

Michael Gasser, Ph.D.

Chris Haynes, Ph.D.

July 27, 1995

Robert F. Port, Ph.D.

Acknowledgements

First and foremost I have to express my gratitude to my advisor, David Leake, without whose support and guidance none of this would have been possible. David's always constructive critiques of my thinking and writing kept me on track and alert to the possibilities and pitfalls of my ideas. He was always a fount of knowledge about people and research I should pay attention to.

I would also like to thank the members of my committee, Chris Haynes, Mike Gasser, and Bob Port, for their interest and invaluable assistance. I greatly appreciate their willingness to offer advice when I needed it and to challenge me to look at my work from a new perspective. Bob Port deserves special thanks for his suggestions about designing empirical tests for ROBBIE and analyzing the results. I would like to include special thanks to George Springer who, while not involved in my research, made me appreciate the value of careful thinking and careful preparation in teaching, qualities that I have also found essential in writing and speaking about my research.

I would like to thank the other members of the CBR reading group for giving me a fresh perspective on CBR: Lincoln Carr, Eliana Colunga-Leal, Phil Doggett, Chuck Shepherd, Raja Sooriamurthi, and Dave Wilson.

Raja Sooriamurthi has been both a friend and a colleague. We have traded advice on our research, and explored conferences together. His friendship has been deeply appreciated.

The CRANIUM lab has been a second home to me for at least three years, and the people who make it up are a special group. I would like to thank them all for making the lab a warm and relaxed place to work, for providing a mountain of useful advice, and for keeping me on my toes during practice talks. The lab includes Doug Blank, Fred Cummins, Doug Eck, Paul Kienzle, Devin McAuley, John Nienart, Cathy Rogers, Raja Sooriamurthi, Keiichi Tajima, and Dave Wilson, plus Lisa Meeden and Sven Anderson, who graduated in earlier years.

I would like to express my appreciation to all the members of the graduate student community in the CS Department for fostering a true sense of community. A complete list of those who deserve mention would fill pages. I would like to thank Kathi Fisler, Liz Johnson, Diana Meadows, and Cornelia Davis for innumerable talks about everything. Knowing a bunch of great women like these has been inspiring to me. I would like to particularly recognize Mike Ashley and Oscar Waddell, with whom I have studied computer science for *nine years*, for frequent Star Trek parties, and advice, assistance, and commiseration whenever I needed it.

My husband, George Calger, deserves my deepest gratitude for his love, support, and encouragement. He listened to my technical babbling with patience, and provided indispensable advice on the proper use of English in my writing. I also owe a deep debt of gratitude to my parents, Robert and Marilynn Fox, for a lifetime of telling me I could do it, letting me do it on my own, and accepting with equanimity when I did do it.

This work was supported in part by the National Science Foundation under Grant No. IRI-9409348. Additional support was provided by the College of Arts and Sciences at Indiana University through a Summer Research Fellowship.

Abstract

A fundamental problem for artificial intelligence is creating systems that can operate well in complex and dynamic domains. In order to perform well in such domains, artificial intelligence systems must be able to learn from novel and unexpected situations. There are many well-researched learning methods for augmenting domain knowledge, but little attention has been given to learning how to manipulate that knowledge more effectively. This research develops a method for learning about reasoning methods themselves. It proposes a model for a combined system which can learn new domain knowledge, but is also able to alter its reasoning methods when they prove inadequate.

Model-based reasoning is used as the basis of an "introspective reasoner" that monitors and refines the reasoning process. In this approach, a model of the desired performance of an underlying system's reasoning is compared to the actual performance to detect discrepancies. A discrepancy indicates a reasoning failure; the system explains the failure by looking for other related failures in the model, and repairs the flaw in the reasoning process which caused the failure. The framework for this introspective reasoner is general and can be transfered to different underlying systems.

The ROBBIE (Re-Organization of Behavior By Introspective Evaluation) system combines a case-based planner with an introspective component implementing the

approach described above. ROBBIE's implementation provides insights into the kinds of knowledge and knowledge representations that are required to model reasoning processes. Experiments have shown a practical benefit to introspective reasoning as well; ROBBIE performs much better when it learns about its reasoning as well as its domain than when it learns only about its domain.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

People learn to cope with the complex situations they face by analyzing their reasoning as well as their actions. Similar benefits may be achieved by monitoring and adjusting reasoning methods in artificial intelligence systems. In this chapter we describe what is required to perform introspective reasoning, our approach to the problem, and the goals of our research.

When people solve problems, whether cooking, taking a quiz, or making a sale, they do not blindly assume that they will be able to find a solution with the first approach they try, nor do they assume that a solution will necessarily succeed. Obstacles often arise both in creating a plan to solve the problem and in executing the solution. Consider the following scenarios:

1. *A person plans a route to drive to a job interview in a new city. As she drives downtown, she discovers her route won't work because a street is one-way the wrong way. She did not check a map for one-way streets in planning her route, and resolves to pay more attention to which streets are one-way the next time.*

2. *A couple decides to make lasagne for dinner. After preparing the sauce and cheese mixtures, they discover that they have no lasagne noodles. One says to*

the other: "Next time, let's be sure we have all the ingredients before *we start cooking."*

3. *A yellow-pages salesperson visits a small business where he finds the owner and his wife. He initially addresses all his comments to the man, assuming that the wife is just a housewife. She asks questions, however, and turns out to be an equal partner in the business. He realizes he could have lost the sale by ignoring her, and he should re-evaluate assumptions about the roles of people in a business.*[1]

4. *A student is taking a math quiz. After spending ten minutes on a problem listed as taking five minutes, she realizes something must have gone wrong, re-evaluates the problem and notices a shortcut she had missed. She quickly solves the problem and reminds herself to consider several alternative approaches to a problem before committing to one method.*

5. *A student writing a C program declares functions for manipulating arrays of characters. After she has spent hours perfecting the code, she looks at the available libraries, and discovers that the functions she defined already existed in a library. She realizes that she should look for already defined libraries before wasting time writing code that may already be provided.*

6. *A novice cook is making rice the first time. He measures the water with a measuring cup, then measures the rice into the same (now wet) cup. As a result, the rice sticks to the cup and must be scraped out. While scraping, it occurs to him that the extra work could be avoided if he had planned to measure*

[1] This example is due to Burke (1993).

the rice first and then the water.

All the previous examples are based on real episodes, and all seem to be natural examples of learning. However, the type of learning that they involve has received little attention in artificial intelligence research. Artificial intelligence research on learning generally focuses on acquiring the domain knowledge needed to perform a task, or on compiling knowledge to speed up performance. However in each of these examples, the reasoners have sufficient knowledge. The problem is in learning how existing knowledge should be accessed and applied.

The scenarios illustrate the fact that people are aware of, and can reason about, not only the actions they are taking, but also the reasoning that led to those actions. They can monitor their reasoning to check its validity, and monitor the consequences of their decisions to see if they are as expected. When people notice something has gone wrong, not only can they reason about how to recover from the failure, but they can also use *meta-reasoning* to learn from it how to avoid the *reasoning failure* that led them astray. In particular, each scenario above illustrates a different aspect of self-monitoring.

In the first two examples, failures while executing plans caused reconsideration of the reasoning process used to create them. The planners' reasoning was at fault, and by identifying what part of their reasoning was flawed, they could learn to avoid similar failures in the future:

1. The driver failed to consider the right features in planning her route: she knew that one-way streets existed, but did not take them into account until she discovered a failure in her planned route. She could learn to avoid similar failures by making sure to note one-way streets in the future.

2. The lasagne cooks failed to include an important step in their plans for cooking: to check that all ingredients are available. They should have altered their reasoning about cooking to incorporate that step.

In the first two examples, the reasoning failure led to a bad outcome: the chosen plans could not be executed successfully. The remaining four examples show that, even if the outcomes of plans are successful, the reasoning process behind them may need to be refined. Noticing the need to improve reasoning for a successful plan requires monitoring performance not only to ensure a successful outcome, but also to ensure that the processes leading to that outcome are as effective as possible. To improve their future reasoning, people must identify places in the reasoning where their performance was suboptimal or unnecessarily risky. Correcting such failures requires meta-reasoning beyond just failure detection; the detected problem may be a symptom of a deeper reasoning failure which may have occurred much earlier in the reasoning process:

3. By realizing that he had misidentified the wife as unimportant, the salesperson avoided a possible failure, but the mistake still pointed to a flaw in how he identified important clients. He could correct it by changing his reasoning about roles to avoid early assumptions.

4. The strategy of the math student was inefficient: she picked one approach to the problem and stuck with it, instead of considering alternative approaches before beginning work. As a result she wasted time, but her poor approach could have led to a correct solution in the end. This scenario demonstrates the need for continuous monitoring of progress in reasoning and in applying solutions: without monitoring her performance, the student would never have discovered

that her approach was not the best one.

5. The C programmer succeeded in completing her assigned task, but wasted time by not considering available libraries first. By including that step in her programming strategy, she could improve her efficiency.

6. The rice cook created a workable but inefficient plan, because he placed the steps in the plan in a poor order. The unexpected need to scrape out the measuring cup led to his realization that he was wasting effort. He could improve his cooking by changing his reasoning to taken more care with the order of ingredients in measuring.

This thesis investigates how artificial intelligence systems can perform the type of learning illustrated in these scenarios: how they can represent their internal reasoning processes with self-models, and use those representations to monitor and reason about their processing and to guide remedial learning to improve the reasoning processes.

1.1 Introspective reasoning in people

By continually re-evaluating their reasoning and its consequences, and explicitly thinking about the reasoning process as well as the actions involved, people are able to discover unexpected difficulties (and opportunities) as they occur, and are also able to learn and improve their reasoning processes over time. This monitoring and self-learning allows people to manage the complexity of the world in which they live: complexity in terms of the enormous amount of information available at each instant, the wide range of possible (and necessary) responses, and the rapidity with which situations in the world change. People can adapt to new circumstances by noticing how their old ways of solving problems are inadequate. When facing a new environment people must often adjust their reasoning and view of the world, for instance: moving from a city with few one-way streets to one with many, or learning how to judge polite and rude behavior in a new culture.

1.1.1 Psychological support for introspective learning

We have given many anecdotes so far which support the idea that people are capable of reasoning about and altering their reasoning methods; further support may be found in some psychological studies (see Flavell (1985), for an overview).

Chi & Glaser (1980) found differences between the reasoning strategies of experts and novices: experts approach problems in a more planful way, spend more time analyzing of a problem before attempting to solve it, and understand better the important features of a problem. They suggest these strategies are learned as part of the process of becoming an expert. Flavell, Friedrichs, & Hoyt (1970) found that older children were better than younger children at monitoring how well they had performed a given task and judging when they had completed it. Younger children predicted they had completely memorized a list of words, then performed poorly. Older children performed as well as they predicted. This shows an awareness of one's own reasoning processes, as well as learning from that awareness to perform better. Kreutzer, Leonard, & Flavell (1975) found that children improved their understanding of their own memory processes as they became older: asked to describe strategies for remembering things, older children tended to describe many strategies and their outcomes, younger children very few. Children appear to be aware of their memorization behavior and learn strategies to help them cope with memory tasks.

ROBBIE's learning centers on refining its criteria for determining the important features of a situation. Psychological experiments also provide evidence that, as children grow, they refine the features that they consider important, and are aware of failures to use appropriate types of features. Gentner (1988) found that younger children depended on surface features of characters and objects in stories when mapping from an old to a new story, whereas older children were able to use deeper features, like the role of the object. Children were presented with a story and toys to represent the characters and asked to act out the story. They were then given different toys which confused the surface features and roles of the characters (i.e., a fish toy might be used to represent a cat character) and asked to act out the same story with the new toys. Younger children (5-7 years old) tended to revert to the objects' physical characteristics, but older children (8-10 years old) recognized the tendency and avoided it, maintaining the *role* of the object over its form. From this she concludes that their understanding of the difference between surface features such as form and deeper features such as role is learned. The awareness of the older children when they started to revert to physical features further suggests monitoring and introspection about their actions.

1.2 Introspective reasoning in AI

If we acknowledge the benefits of introspective monitoring and correction of reasoning methods for people, then we must acknowledge that similar benefits apply to the decision-making and problem-solving of artificial intelligence systems.

Artificial intelligence systems are increasingly designed to operate and interact with complex worlds which are knowledge-rich and dynamic[2]. When a system is situated in a rich world, informing the system ahead of time of all possibly relevant information and reasoning methods is intractable. It is extremely difficult to anticipate the situations the system will see, what information will be important, and what response circumstances will demand. If in addition the world contains many uncontrolled variables, the task of fixing knowledge and reasoning requirements becomes impossible. To develop systems which can successfully interact with real-world or complex domains, we must explore ways of making systems more flexible, more adaptable, and more sophisticated in their approach to each situation: more able to *learn* what they need to know from their experience.

In terms of handling domain complexity, systems which can learn by adding to their knowledge of their domains are a step above those systems whose reasoning methods and knowledge have been completely pre-determined by their designer. Learning new domain knowledge allows a system to improve its understanding of its domain, and to adapt somewhat to new circumstances. The work of the system designer is simplified by not having to select a priori all the knowledge ever required for the system's tasks. Many systems have been designed which improve their knowledge of the world and of how they operate in it (e.g., (Michalski et al., 1983, 1986; Kodratoff & Michalski, 1990; Michalski & Tecuci, 1994)). However, most learning systems are incapable of changing the processing or reasoning methods they use to manipulate their world knowledge. They may learn new facts or new solutions, new information about the domain in which they operate, but they do not learn new ways

[2]By knowledge-rich we mean to exclude domains whose knowledge is easily and concisely encapsulated, such as the typical "blocks world" domain. By dynamic we mean that the task or domain involves elements which are outside the reasoner's control: the world may change without any action by the system itself.

to access or manipulate those facts.

Acquiring knowledge is a well-known problem for artificial intelligence systems. Approaches developed to solve the knowledge acquisition problem include verification of correct knowledge using a set of test problem and correction by a human expert, case-based learning — sometimes expert-guided, and explanation-based generalization of knowledge (Marcus, 1989). All these methods only address the learning of domain knowledge; they assume if a failure occurs that the problem stems from faulty knowledge, not from a failure or the reasoning to apply the knowledge correctly. Our approach, by contrast, is intended to extend the system to discover reasoning failures and learn from them to reason better.

Just as systems which do no learning at all have problems in complex domains because of the difficulty of selecting and fixing all the knowledge ahead of time, systems which can augment their world knowledge, but which can never learn new ways of processing that knowledge are also restricted by the limitations of their fixed reasoning methods. Such systems must always assume that their domain knowledge is at fault when something goes wrong, and cannot consider that the reasoning that produced a correct answer might be faulty.

In a domain whose complexity makes determining the necessary world knowledge difficult, it will also be difficult to determine the correct reasoning approach to manipulate that knowledge effectively. A particularly important problem which this research addresses is determining what features of a situation in a knowledge-rich domain are relevant to the solution of a given task. If the circumstances in which a system operates change, the system must be able to adapt both its knowledge and its reasoning to fit the new circumstances. Thus artificial intelligence systems in knowledge-rich and dynamic domains need the capacity for introspective reasoning and learning.

Other systems have acquired meta-knowledge about how to apply their domain knowledge. SOAR can create "meta-rules" which describe how to apply rules about domain tasks and acquire knowledge (Rosenbloom, Laird, & Newell, 1993b, 1993a). Meta-rules are created by chunking together existing rules. Chunking is the only way for learning to occur; a full introspective reasoner should be able to alter its reasoning in multiple ways (e.g., by deleting or altering rules). The introspective process in SOAR is driven by *impasses*, points where the processing cannot proceed normally; SOAR cannot learn from failures which produce successful but suboptimal results. Furthermore, as for the general approaches described above, SOAR assumes that the processes for selecting rules and carrying out their instructions are infallible; ROBBIE models all of its underlying reasoning processes.

ASK learns "strategic knowledge" rules which described what reasoning actions to take under given circumstances (Gruber, 1989). While this is close to the kind of knowledge our approach will learn, learning in ASK is guided by a human expert who provides all the knowledge about the reasoning process. Our task is to develop the means to represent that knowledge about reasoning within the artificial intelligence system, and to have diagnosis and repair of reasoning performed without the intervention of a human expert.

1.3 Requirements for introspective learning

In order for a system to reason about and evaluate its reasoning methods, it first must have access to information about its reasoning: what reasoning processes it has applied up to the current point and what it is currently applying. The system must maintain a trace of its reasoning process which describes the sequence of decisions it has made and the knowledge structures it used or created in the process. Such a trace gives the system access to its past and present reasoning history, but it also needs the means of determining from that information whether faulty reasoning has occurred and how to correct it.

We must determine what kind of knowledge of the reasoning process is required for introspective diagnosis; we must also find reasonable restrictions on the scope and detail of that knowledge, as we cannot assume the system has access to perfect knowledge of what its reasoning should have been. If the introspective knowledge were perfect, then that perfect, detailed knowledge would be available to solve the original task in the first place, and no reasoning failures would ever occur. Our model of meta-reasoning, then, must be a limited abstraction of the actual reasoning processes of the system.

We can break down the problem of noticing and correcting faulty reasoning into four parts:

1. having criteria for deciding when to check for faulty reasoning,
2. establishing from those criteria whether or not a reasoning failure has occurred,
3. determining the ultimate cause of a detected failure, and
4. altering the reasoning process to avoid similar future failures.

Having criteria for when to check for faulty reasoning: The first requirement above is to have criteria — in the form of knowledge — that describe the desired reasoning behavior of the system and outcomes of the system in order to judge whether the system's reasoning at any given point seems correct or not. These criteria must produce *expectations* about the reasoning behavior the system should exhibit. These expectations may be explicitly represented in the introspective knowledge of the system, or they may be implicit in the reasoning system.

The criteria for judging the performance of the system's reasoning may include expectations about different portions of the system, task, and domain. There may be expectations about the reasoning itself — "If this reasoning rules is applied, this new intermediate value will be produced," — or about the solution that the reasoning produces — "When I approach a turn I have planned, I will be able to make the turn." Expectations may also exist which refer to features of the domain apart from the reasoning system itself: "Streets will all have at least two lanes." Expectations may also vary in their level of generality and scope: for example, a specific expectation might be "I will be able to turn right onto 8th street", versus a general one whose scope is the entire problem, like "The route I've planned will get me to the office in time."

Establishing whether a reasoning failure has occurred: To tell whether a failure has occurred, the system must have the means of determining which expectations are currently relevant to the reasoning process and which expectations may be verified by current observations. A reasoning failure is possible whenever expectations about the system's desired reasoning performance fail to be true of the actual reasoning.

It is important to make a clear distinction between *expectation failures*, *reasoning failures*, and *outcome failures*. An expectation failure is, at root, any event which was not predicted by the system; such an event may not indicate an outcome failure where the system's main task fails, but could be an unexpected *opportunity*. Independent of whether an expectation failure is an unexpected failure or success for the task, it may be the result of a reasoning failure, if the system should have known the event would occur and have planned for its existence. However, not all expectation failures result from reasoning failures: unexpected events which are based on knowledge that the system simply does not possess are unavoidable. For example, if a street is closed due to a traffic accident, it may be an expectation failure and an outcome failure, but not a reasoning failure, as no improved reasoning could have predicted a traffic accident at that particular time.

We distinguish two classes of expectation failures to be detected. The first and easiest to detect are those failures which are catastrophic: failures which make it impossible to continue with the current problem solution. The lasagne and route planning scenarios above are examples of this type of failure. To detect catastrophic failures the system must have expectations about the actions involved in the solution of the problem. The second class of failures are those which involve inefficient processing either in the creation or application of a solution. The math student is one such scenario, as is the rice cook. Detecting the second class of failures is more difficult; it requires expectations about the reasoning process itself, expectations about how long something should take, how many steps should be involved, and how to tell if proper progress is being made (see also Fox & Leake (1994)).

Determining the ultimate cause: Once the system has detected an expectation failure, it must be able to determine what point in the reasoning process the detected failure was created. This task is perhaps the most difficult part of introspective evaluation of the reasoning process, because there is no guarantee that the system will detect every reasoning failure at the instant it occurs. In fact, the scenarios described earlier indicate quite the opposite: very often reasoning failures persist *unnoticed* until very much later in the overall reasoning task. The system must be able to discover causal relationships between particular criteria at different points of the reasoning process in order to explain where a detected failure was generated.

Altering the reasoning process: The requirement that introspective reasoning be able to repair reasoning flaws means that the system must have the ability to alter its own reasoning process. It must therefore have access to the process and must be able to decide what change to make on the basis of what went wrong. If we assume that the system will have pinpointed the ultimate cause of a detected expectation failure before attempting to generate a repair, the task of deciding what to change is not as daunting. For a particular point in the reasoning process, the alterations which could reasonably be made are rather limited. Considering the rice-cooking scenario above, the detected problem of having the rice stick to the measuring cup could be traced back to a failure to anticipate the interaction between water in the cup and rice in the cup, which leads to a repair which says, for the future, to anticipate that rice will stick after water, so measure the rice first.

To sum up, a system must have access to knowledge about its reasoning process up to the current point in order to be able to detect and explain reasoning failures. It must also have implicit or explicit expectations both about the results of its reasoning in its domain and about the internal reasoning process itself. It must be able to detect expectations failures during the reasoning process and during the execution of the problem solution. The system must be able to explain an expectation failure in terms of underlying reasoning failures, and must be able to determine how to alter the reasoning process to correct the failures for the future.

1.4 The ROBBIE system

We have implemented in ROBBIE (Re-Organization of Behavior By Introspective Evaluation) a system which combines case-based reasoning with a separate introspective component using a model of the ideal reasoning methods of the case-based system itself. This system serves both as a testbed for our theoretical ideas about modeling for introspective purposes and as a demonstration of the benefits of using an introspective model such as we propose. ROBBIE's performance task is to generate plans for walking from place to place on a set of streets, given limited knowledge about the map and a few initial sample routes. It examines the quality of those plans through execution in a simulated world. As a *case-based* system ROBBIE learns by adding new cases (in this case, new route plans) to its case memory. Adding cases alters ROBBIE's world knowledge: as the case memory grows it reflects a better and better understanding of the map in which the robot moves. In parallel with the case-based reasoning process, ROBBIE's introspective component monitors the reasoning process of the planner and compares the actual processing to its expectations of the ideal performance of the case-based reasoning process. When expectation failures occur (either because of catastrophic or efficiency failures) the introspective component of ROBBIE can suspend the planning task while it attempts to explain the failure and repair the system. Under some circumstances, the *performance task* (planning and executing plans) may continue until enough information is available to assess the failure.

ROBBIE's introspective model describes the reasoning process of ROBBIE's planner from start to finish, and the introspective reasoner monitors and evaluates the entire process, but we have focused on only one important and powerful repair strategy: altering the features used in retrieving old solutions. Failures resulting from other problems than poor retrieval will be detected and explained, but not repaired. We expect to extend the repair component to incorporate alterations to other portions of the planning task in the future. Modeling the entire reasoning process is required for detecting failures related to just this repair strategy; a failure introduced at the beginning by a poor retrieval might not become evident until the plan had been adapted and was being executed, or even later.

Reflection: It is important to note that while the approach we advocate does involve the use of introspection to permit a system to examine and evaluate its reasoning process, our approach is not a *reflective* one. In order to be considered fully reflective, the process performing the base level task must also be able to manipulate and reason about its own process, and alter its own processing behavior (Ibrahim, 1992). In our approach, representation and manipulation of the main process are performed by a separate reasoning process, the "introspective reasoner," which uses a different form or representation than the case-based reasoning system underlying it. Our approach uses a separate reasoning component which represents the underlying reasoning process in a different manner than the representation used in the case-based component, and examines *artifacts* of the underlying process, not the actual code in which it is implemented.

The goal of our introspective reasoner does not require access to the actual code with which the reasoning system is implemented, or the ability to generate arbitrary changes to that underlying reasoning system, but rather requires a model which represents expectations about the *ideal* reasoning the underlying system should exhibit. For our purposes, artifacts of the actual processing are sufficient and a full reflective system, at this point, unnecessary and more difficult to implement. Nevertheless, incorporating reflection to the point of introspective reasoning about the introspective reasoner, and using similar mechanisms for both base level and introspective tasks, remain future possibilities.

1.5 Results

We can evaluate the introspective reasoning framework we have developed at several levels. First we can consider what ROBBIE has shown about the kind of knowledge needed for introspective reasoning, and the kind of model that is required to balance failure detection and failure explanation and repair. We have found in our research that introspective knowledge must span a range from very specific to very general: that it must provide expectations at a low level for detecting failures and determining specific repairs, and must provide expectations at a high level in order to facilitate the process of explaining failures and tracing from detected failures to root causes. The model must be structured to assist failure detection and explanation of failures in different ways. The model must also be designed to be easy to modify for different underlying reasoning systems by incorporating general mechanisms and vocabulary for representing the model, as well as a highly modular structure for the model.

In constructing a working hybrid of case-based learning and introspective learning, we enable empirical testing of introspective learning's promised advantages, and we evaluated ROBBIE in action to verify that introspective reasoning produces the learning we expect of it. We will describe selected examples of ROBBIE's reasoning to show its performance under a variety of situations.

We will also evaluate ROBBIE's performance to verify that the changes made by introspective reasoning do improve ROBBIE's performance over time. We chose to evaluate ROBBIE's performance with and without the introspective component, under circumstances of relative difficulty for both case-based learning and our form of introspective learning. From these experiments we can judge the extent of the improvement introspective reasoning provides, and under what circumstances its benefit is maximized. These experiments demonstrate a benefit for learning new retrieval indexing features, driven by introspective failure detection and explanation.

1.6 Goals of this research

We claim that introspective reasoning for improving reasoning processes will improve the performance of an overall learning system in a complex domain. In particular, the ultimate goals of this thesis are

- to develop an approach to introspective reasoning that provides equally for monitoring to detect failures and explanation of detected failures as equally important to the introspective reasoning task;
- to create a framework for constructing introspective reasoning systems for application to a variety of underlying reasoning systems;
- to implement and test this framework with an underlying learning system;
- and to develop methods for empirically evaluating systems which combine introspective reasoning with other reasoning methods.

In this section we will elaborate on these points.

1.6.1 A framework for introspective reasoning

We described four requirements for an introspective reasoning system in Section 1.3. Here we will consider each requirement in turn and sketch how we have chosen to approach it. Our general approach uses a model of the system's own reasoning processes to detect *and* to explain reasoning failures (see also Fox & Leake (1995c)). Using model-based reasoning to represent the reasoning task itself was proposed by Birnbaum et al. and implemented by them for self-debugging planners (Birnbaum, Collins, Freed, & Krulwich, 1990). They suggested that model-based introspective reasoning could be applied to case-based reasoning systems (Birnbaum, Collins, Brand, Freed, Krulwich, & Pryor, 1991). However, their ideas were never implemented for case-based reasoning, and their approach has never been empirically evaluated. This research builds upon their suggestions for case-based reasoning, and empirically evaluates the resulting framework. One goal of this work is to study how to represent case-based reasoning for introspective purposes, both in terms of general case-based principles and in terms of our particular implementation.

Criteria: To implement criteria for telling if a failure has occurred we used a model of the ideal reasoning of the underlying system. The model contains explicit expectations about the results of each point in the reasoning process in terms of both later reasoning and domain-level actions. The model contains modular clusters corresponding to the actual structure of the underlying reasoning system, and contains a hierarchy of expectations at differing levels of specificity. For example, a specific-level expectation about the retrieval mechanism of a case-based reasoning system might say "Retrieval will discover at least one relevant case."

Detecting failures: Expectations in the model are compared to the actual reasoning performance of the system to detect discrepancies. Discrepancies between the expected, desired behavior and the actual behavior are expectation failures. The portions of the model used for detecting failures are specific and limited in scope; determining whether a failure has occurred means considering the set of specific expectations about the current point in the system's reasoning. Monitoring for expectation failures occurs in parallel with the system's reasoning process: as each step in the reasoning process occurs, expectations relevant to it are selected and considered. For example, the specific-level retrieval expectation above would be evaluated at each step of the retrieval process. If no cases were retrieved, the above expectation would fail.

Explaining failures: The model used to represent the criteria for reasoning failures is also used to search for explanations of detected failures, by finding an earlier expectation that was at fault. The model incorporates causal information relating one expectation to another, and a path is traced from the detected expectation failure to a "root cause" expectation that may have failed and gone undetected. The system must be able to pinpoint a root cause in order to suggest a specific correction to the reasoning process which would prevent the failure in the future. For the sample expectation failure in the previous paragraph, an explanation might trace to a previously undetected expectation failure: the planner is expected to use the best retrieval criteria.

Repairing failures: Repairs to the reasoning process are attached to particular expectations in the model, so that when that expectation fails, the attached repair is suggested. The description of the repair in the model is not enough to specify exactly how and what to change, so the system includes mechanisms for determining how to implement the repair. The repair module takes the description of the explained failure and the suggested repair strategy and uses that information to fill in the details of an actual repair and to make the actual alterations. A general repair module would have a set of different repair strategies and the means of implementing them; however, we have chosen to focus on one very important strategy: adding to the set of features to be considered explicitly in constructing a solution. For the example above, recognizing the failure to use the right retrieval criteria might require knowing what criteria should have been included. This knowledge leads to an obvious repair: make the retrieval mechanism use the missing criteria.

Our approach incorporates into the structure and content of the model a balance between the needs of failure detection and the needs of failure explanation. We have considered carefully the kinds of knowledge and expectations that are required for introspective reasoning to succeed. We have addressed the question of the level of abstraction of the expectations in the model in order to describe the reasoning process as a whole, and still include specific information required for ease in detection and repair.

1.6.2 General applicability

The issues we are exploring have a wider applicability than this particular implementation, planning alone, or case-based reasoning alone. The framework we have developed for implementing an introspective reasoning system is designed to be re-used for reasoning about other systems. Because the model is declarative, it may be modified without difficulty, and uses representations and structures which are system-independent. The mechanisms for communication between the introspective reasoner and its underlying reasoning system provide a generic format for integrating introspective reasoning with existing artificial intelligence systems. We have examined in detail the classes of knowledge needed to detect and repair reasoning failures: such knowledge needs will remain the same regardless of the way in which introspective reasoning is implemented.

Our introspective framework could be applied to monitor and improve the reasoning of systems in many domains where we can specify what the best performance should be. Domains which could benefit from introspective reasoning include: robot control, scheduling for many objects or other variables, autonomous agents for organizing information or information-gathering (i.e., an "intelligent" web searching system), and many others. Each of these domains is complex and highly dynamic, the perfect environment for a system that learns to improve its reasoning as well as improving its domain knowledge, when experience indicates the need for improvement.

1.6.3 Evaluation of the model

The combination of introspective-level learning with ordinary domain-level learning is relatively new, and has had, so far, little empirical evaluation to examine its effects and compare them to the hypothesized results. We have chosen one method of empirical evaluation to test ROBBIE: comparing the performance of ROBBIE with case-based learning and introspective learning against ROBBIE's performance when introspective learning is disabled (see also Fox & Leake (1995a)). We must ensure that the tests involved cover as wide a range of possible situations as possible to ensure that introspective learning improves performance across the board, and not just in a small range of situations. In addition, examination of these differing circumstances can illuminate how the performance of introspective reasoning is affected by the order of the situations to which it is exposed (see also Fox & Leake (1995b)).

1.6.4 Ultimate goal

The ultimate, long-range goal of this research is the development of an approach to learning that uses the same set of mechanisms for learning about its knowledge and mechanisms at all levels; learning domain knowledge, and learning by improving all of its reasoning processes. A system implementing such an approach would ideally be reflective, having the ability to bring any of its mechanisms to bear on any particular reasoning it has to do (whether about its task or itself). At this point such a system is long in the future, but we will address how our current work is a step in the right direction, and what issues must still be considered.

One issue we will discuss in Chapter 8 is the feasibility of scaling up our approach to more complex domains and tasks. Our current implementation operates in a rich but restricted and idealized domain. The knowledge of the system may be limited and interesting performance still be produced. We must analyze the cost and complexity of our approach in order to extend it to real-world applications or large-scale problem domains.

1.7 Outline of the following chapters

In Chapter 2 we describe other research related to ROBBIE, forming the basis for our work or providing an alternative approach. We describe case-based reasoning in detail, as well as model-based reasoning for diagnosis. In Chapter 3 we provide an overview of ROBBIE and its domain, describing its simulated world and sketching how the various components fit together. In Chapters 4 through 6 we describe the planning component of ROBBIE and the introspective component in greater detail. We follow that in Chapter 7 by describing the experiments performed to evaluate ROBBIE's performance. Finally, in Chapter 8 we describe our conclusions about the issues our research has raised, including future directions for this research.

Chapter 2

Background

> Our research builds on the case-based reasoning paradigm, and is part of an upsurge of interest in representing reasoning processes. We describe the approaches underlying our work and the similarities and differences with other research on related problems.

Our research on ROBBIE is relevant to three main areas of artificial intelligence: case-based reasoning, reactive planning, and introspective reasoning. ROBBIE's planner incorporates both case-based and reactive planning to create and execute route plans and refines this process by introspective reasoning. In this chapter we describe the fundamental tenets of case-based reasoning and how ROBBIE's planner relates to other case-based planning systems. We describe other work relating to the problem of index refinement which ROBBIE addresses. We discuss the variety of approaches to reactive planning, other work which has integrated plan creation with reactive execution, and other planning systems which address some of the same issues ROBBIE addresses.

The focus of our research is not on the planning aspects of ROBBIE, per se, although ROBBIE's planner does incorporate several interesting innovations, as we will discuss in Chapter 4. We are most interested in examining the introspective aspects of ROBBIE: how introspective knowledge may be represented, how it may be structured to support its uses, and how it integrates into the system as a whole. We therefore concentrate on the introspective frameworks to which ROBBIE's introspective reasoner is related, and illustrate ROBBIE's contributions by comparing its introspective framework to other approaches to the task of introspective diagnosis and repair of reasoning failures.

2.1 Case-based reasoning

Creating artificial intelligence systems that reason by recalling previous situations stems from the realization that people often use past experiences when facing new problems (Schank, 1982). A student solving a math problem might recall a previous problem with similar features and try to apply the method that worked to solve the previous problem, altering it to fit the new situation. Recalling the right previous situation can make responding to a new situation easier and response more rapid; a person can re-apply a method worked out previously for solving a problem without re-creating the reasoning from scratch. A number of psychological studies provide support for the importance of this problem-solving process in human reasoners (see (Ross, 1989) for a summary).

Using past experience as a jumping-off point for current problem-solving has a number of advantages for artificial intelligence systems. The cost of creating a solution from first principles may be avoided by re-using the reasoning implicit in a solution to a similar problem. Using existing solutions also avoids the need for a strong and complete theory of a given domain; important features of the domain which may not

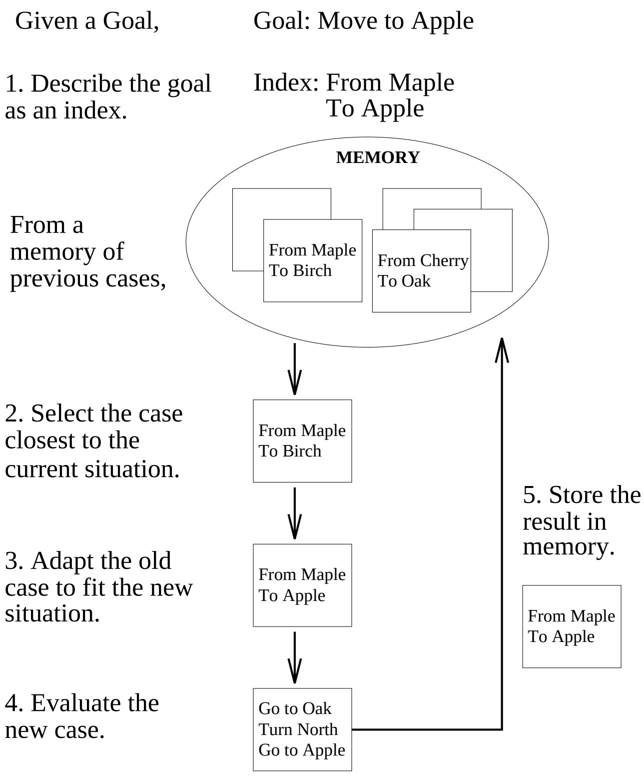

Figure 2.1: A typical case-based planner

be apparent will exist implicitly in successful solutions in that domain. For example, it is not necessary to understand the chemical processes involved in cooking to adapt a recipe using veal and broccoli to one using beef and a mixture of vegetables.

The case-based reasoning process can be broken into separate tasks, although exactly where the lines are drawn between tasks depends on the particular application. The process may be generally divided into five components: describing the problem, retrieving the best matching case(s) in memory, adapting those cases to apply to the current situation, evaluating the resulting solution, and recording the new solution in memory for future use. Some CBR systems focus on a few aspects of this set, ignoring others or leaving their solution to a human user of the system. Figure 2.1 illustrates the typical case-based reasoning process for a case-based route planner.

Describing the problem: In order to select the "best" match in memory, the current situation must be described in terms which permit a comparison to existing cases. The salient features of a situation must be collected or derived from an initial description to form an *index*: a set of features which may be matched against the same sort of features for cases in memory.

Retrieving cases: To retrieve an appropriate case, a CBR system compares the index describing the current situation to the indices of cases in memory, and judges how similar the two cases are from the similarity of their indices. Similarity should correspond to "adaptability:" the best case to retrieve is the one requiring the least effort to fit to the current situation. The index should embody those aspects of a situation which indicate adaptability and the methods for assessing similarity should choose between cases to appropriately select the most adaptable case.

Adapting cases: Most of the time the case or cases retrieved from memory will not be perfect matches for the current situation. Adaptation is required to alter those portions which are incompatible with the new problem. For instance, the SWALE system selected a case describing Jim Fixx's death in an attempt to explain the death of the racehorse Swale. Since horses don't jog recreationally, the explanation of Jim Fixx's death must be altered to apply to racehorses (Schank & Leake, 1989). Ensuring that the adaptations performed maintain the integrity of the original case and successfully apply to the current problem makes adaptation of cases a key issue for CBR systems. As a result, some systems avoid adaptation by leaving it to a human user. ROBBIE, however, does perform adaptation of its planning cases.

Evaluating solutions: Adaptation is often a heuristic process for case-based reasoning systems. The resulting solution may have hidden defects stemming from information that the system did not possess or incorrectly applied. Evaluation of adapted solutions is therefore an important part of the overall process, to determine the validity of changes made. Evaluation methods differ: human feedback, feedback from a separate computer system, simulation of applying the case, or actual application of the case.

Learning cases: The storing of cases is important as the phase that incorporates learning into the case-based framework. Once a successful solution (or an instructive failure) has been found, adding that solution as a case to memory enables the system to extend the breadth of situations it can successfully handle. Each case may be an incremental extension to new knowledge: the explanation for Swale's death might help explain the death of a famous greyhound. The case storage process depends upon a successful encapsulation of the situation to be stored, and creation of a correct index describing its salient features.

2.1.1 Planning in the context of CBR

Case-based reasoning has been applied to the tasks of explanation, understanding, diagnosis, tutoring, and many others, but case-based planning has been a popular task, perhaps because of the immediate crucial importance of planning for practical applications (e.g., (Fowler, Cross, & Owens, 1995)). Where a case for an understanding system might describe a scene or set of events, a case-based planner's case is likely to be a set of instructions for achieving some goal.

ROBBIE is most closely related to the planner CHEF in its structure and approach, although the domains to which each is applied are very different. CHEF creates recipes for stir-fry (and other) dishes. Like ROBBIE, CHEF conforms closely to the sequence of tasks described above. CHEF uses an external, sophisticated evaluation system to determine whether a new recipe is a success. ROBBIE performs evaluation itself by integrating execution of its plans into the planning process. When a recipe plan created by CHEF fails, CHEF learns the interactions among ingredients or cooking steps that cause the failure. A description of the causal sequence leading to the failure is provide to it the separate evaluation system. It then augments its ingredient knowledge or the index used for the new case to represent the previously absent interactions. ROBBIE alters its indexing, but uses an analysis of its reasoning process to detect a broader class of failures: its introspective reasoning allows it to detect a bad retrieval even when its case adaptor can repair the problems to produce a successful solution.

Other case-based planners include PLEXUS (Alterman, 1986) which applies cases describing routine, everyday plans for traveling by public transportation. PLEXUS is concerned with the integration of planning and action, as ROBBIE is, and performs adaptation of retrieved routine plans as an execution task. A plan step which cannot be applied immediately is replaced by an alternative, chosen on the basis of the current external situation. ROBBIE uses a more traditional adaptation process to form a high-level plan, then uses reactive execution to complete the adaptation in interaction with the external world.

ROUTER is another case-based *route* planner, which creates routes for navigating a college campus (Goel, Callantine, Shankar, & Chandrasekaran, 1991). ROUTER combines case-based plan creation with a more traditional model-based approach.

Unlike ROBBIE, it is not concerned with execution of plans or dynamic changes to the world which may suddenly invalidate an previously perfect plan. ROUTER has also been combined with an introspective reasoning component, Autognostic, (Stroulia & Goel, 1994) as we will describe in Section 2.4.

ROBBIE differs from all these case-based planners in its focus on incorporating introspective reasoning into the system to detect and repair failures due to faulty reasoning. The combination of case-based planning, to create a plan outline, with reactive execution, to fill in the details, is also unique.

2.1.2 Determining features for indexing criteria

Within the case-based reasoning paradigm, multiple methods have been proposed for determining relevant indices. Explanations are often used to determine the relevance of features when assigning indices to new cases. CABER provides plans for recovering from machine failures of a milling machine to a human user (Barletta & Mark, 1988). New repair plans are stored by deriving the important indices from explanations based on knowledge of the machine. in question. Leake & Owens (1986) determine the acceptability of a case for explaining an anomaly by characterizing the anomaly and the goals of its explanation. In AQUA, new cases are similarly indexed by explanation-based generalization of the "stereotypical" features of situations, and the objects and characters in them (Ram, 1993). However, less attention has been devoted to the central questions addressed by ROBBIE: when and how to alter the indexing criteria based on an *existing case* in memory.

Some approaches alter indices in response to external feedback. CELIA initially selects cases using a wide range of possible features (Redmond, 1992). It prunes the features for a case when feedback from a human expert indicates the case is inapplicable. It also alters the indices to retrieve a case which was incorrectly omitted. Veloso & Carbonell (1993) use *derivational analogy* to produce solutions by examining the stored reasoning traces of previous solutions, in PRODIGY. PRODIGY's memory manager suggests cases to its problem-solving component, and alters the indices for the retrieved cases based on positive or negative feedback from the problem-solver on the utility of the suggested cases. ROBBIE, by contrast, uses introspective analysis of its reasoning performance to determine whether new indices should be learned.

IDEAL investigates index learning in the domain of device design by using a model of the structure and function of a device to determine the important structural features of a given device description, for indexing that description as a case (Bhatta & Goel, 1993). In CADET, Sycara & Navinchandra (1989) alter the indices used to retrieve cases for designing new devices by applying operators which elaborate the important features describing the new desired device. The learning of new indices is driven by design requirements, not failures of the existing index criteria.

In addition, when ROBBIE chooses new indices, it uses knowledge of both the faulty retrieval and the case that should have been retrieved to guide the choice of new indices. This allows it to select indices that are useful for discriminating between the cases currently in its memory.

2.2 Reactive planning

Reactive planning grew out of a reaction against "armchair" deliberative planning systems[1] create plans never tested by actual execution, because they assume the planner's knowledge of its domain is perfect, and nothing in the domain changes

[1]Deliberative planning refers to systems which plan out the entire problem before executing any part of the plan.

except when caused by the planner. Many planners depend on the assumptions that the plans created will never go wrong, that no other actors exist in the domain except the planner itself, or that alternatives to every conceivable problem may be incorporated in a single plan. Creation of a plan that may be executed blindly seems unrealistic for planners operating outside carefully controlled domains. Assuming that no other actors exist is obviously limiting. In addition, the time taken to construct elaborately detailed plans might mean that the situation in the world has altered and the plan is obsolete.

Reactive planning, by contrast, focuses on the conjunction of moment-to-moment decisions and actions to lead eventually to the goal. Reactive planners tie their decisions closely to input about their world and to momentary actions taken. Reactive planning is especially valuable in situations, like navigating a crowded room, where a quick response is required and a complete plan would become obsolete before execution could take place. A fully deliberative planner operates under the credo "A stitch in time saves nine," a reactive planner by "He who hesitates is lost."

ROBBIE incorporates both deliberative (in the form of case-based planning) and reactive planning to gain the benefits of thinking ahead and responding to immediate feedback. Its reactive component is based on the Reactive Action Packages (RAP) model of Firby (Firby, 1989). Firby implemented his RAP model for a system which simulates a delivery truck responding to the commands of its dispatcher and the needs of its current situation.[2] Firby' system focuses on the problem of opportunistically-managed multiple goals, as it may have immediate needs (such as being low on fuel) and several outstanding delivery goals, at the same time. From its current goals, both

[2]Freed & Collins (1994a) uses the RAP model to underlie the introspective reasoning approach in RAPTER, as we will discuss below.

internal and external in source, the reactive planner selects a RAP which describes a range of appropriate immediate responses. One response is selected based on its applicability to the current situation. Applying the right method for a given context means taking some actions in the world, or adding to the set of goals waiting to be fulfilled. After a time step, the world situation is re-evaluated an another RAP may be selected to respond to the changes since the previous time step.

ROBBIE's reactive component differs from Firby's system because of its combination with a deliberative planner. ROBBIE's case-based planner produces high-level plans, which guide the reactive component much more than Firby's planner is guided by its goals. Firby examines issues in the management of multiple goals; ROBBIE has only one goal at a time (unless one considers goals at different levels: a goal to reach the goal location, a goal to finish executing the current plan step, and a goal to take particular actions in the world). ROBBIE's reactive planner is also implemented using case-based reasoning: *planlets*, ROBBIE's equivalent to RAPs, are stored in the case memory and retrieved by indices describing the current situation in the world.

Firby's RAPs incorporate both high-level and low-level actions. Other approaches to reactive planning focus on increasingly low-level actions and an even closer link between observations and actions.

The Pengi system plays the video game Pengo (Agre & Chapman, 1987). At each moment, Pengi selects a *routine*, an action to perform, based on the current situation. Local objects are labeled according to their function from Pengi's perspective (i.e., the-bee-that-is-chasing-me) and re-labeled in the next moment. Arbitration between competing routines on the basis of priorities hardwired into Pengi by the designer; thus Pengi cannot learn when to perform certain actions. By using compiled routines chosen with hardwired priorities, Agre and Chapman avoid any explicit reasoning or use of symbols for objects by the system. However, it is difficult to see how Pengi would achieve a longer-range goal, requiring coordination of many actions; for instance, in the case of Pengo, the goal of the game requires moving special objects from positions scattered across the board into a particular configuration.

Meeden (1994) takes a very different approach to planning and action in Carbot. Meeden trains a recurrent connectionist network using reward and punishment reinforcement learning to control a robot in order to achieve simple goals. For instance, Carbot must learn to alternatively approach and move away from a light source, or to keep moving without hitting a wall. Complex behaviors emerge from low-level association of sensors and actuators, mediated by the connectionist network. Meeden's thesis is that from such simple, *learned* behaviors more sophisticated behavior may emerge which would be classified as "planful." How far this approach may go remains to be demonstrated. ROBBIE, by contrast, assumes explicit deliberative planning, in combination with a representational approach to reactive planning.

At the extreme end of the reactive planning spectrum, and perhaps unwillingly classified as such, is Brooks' subsumption architecture (Brooks, 1987). The "planning" aspect is nearly missing completely; Brooks ties sensor input and action together without an explicit representation of input or action intervening. The choice between actions is mediated between different hardwired processes performing different functions which map from input to action. For example, one process might encourage wall-following behavior of a robot, and another might investigate objects. One process might override another in some circumstances, or their recommended actions might be combined to produce a compromise behavior. While this is an interesting approach to low-level control and navigation, we do not feel it captures the abstract reasoning required for long-range planning, which people perform when preparing to travel to a new location.

2.3 Integrating deliberative and reactive planning

One goal for ROBBIE was to address the conundrum faced by a system that must plan ahead, but which operates in a domain where the situation changes dynamically. ROBBIE addresses this by combining deliberative and reactive planning, and by adding new features of situations through introspective learning as they turn out to be important. Other work has addressed the integration of deliberative and reactive planning: Gat (1992) implements a framework based on Firby's RAP system in ATLANTIS, in which the reasoning process can request advice from higher-level deliberative planning processes. Nourbakhsh, Powers, & Birchfield (1995) describe an alternative approach that maintains a set of the possible locations in which a real physical robot might be, and re-plans whenever the most likely possibility proves to be false. This approach demands a very complete understanding of the robot's world. Maintaining a list possible locations may be infeasible when planning for a robot navigating a city instead of an office. Downs & Reichgelt (1991) combines reactive planning with higher-level planning, using multiple levels which apply different planning approaches, culminating in a STRIPS-like planner at the top level. Unlike our research, these systems perform only limited learning even about their domain, and none are concerned with learning to improve the system's reasoning mechanisms.

2.4 Introspective reasoning

Our research has focused, not on issues in planning, but on issues of introspective reasoning. Recently, there has been a surge of interest in systems which have explicit knowledge about reasoning methods, and which can use that knowledge to diagnose and repair their own reasoning methods, or understand the reasoning of others (Cox & Freed, 1995). The introspective reasoning framework we have designed performs diagnosis and repair of its own reasoning, but its approach to the representational needs for modeling reasoning appears useful for these other introspective tasks as well.

In this section we describe the research upon which ROBBIE's model is based, and compare ROBBIE to other approaches to introspective reasoning for diagnosing reasoning failures. We also discuss other approaches for improving reasoning methods.

2.4.1 Model-based reasoning for introspective self-diagnosis

Our approach to introspective reasoning derives from the proposal of Birnbaum et al. (1991) to use model-based reasoning to represent expectations about the reasoning process of a case-based planning system. The framework they propose forms the basis from which ROBBIE's model was developed. Like ROBBIE's, their model is made up of assertions which form expectations about the ideal underlying reasoning process. Expectations about the performance of the planner, separate from the model, are examined directly to detect failures. Such performance expectations are linked through *justification structures* to the assertions in the model which describe the assumptions upon which the expectations depend. The explanation of a failed expectation, in Birnbaum et al.'s model, is a process of tracing back through these justification structures, to determine a series of assumptions that might be faulty leading to a repairable faulty assertion. Their proposed model for case-based planning does not specify the level of detail of the assertions in the model, but describes only highly abstract expectations. The justification structures form the link between abstract expectations and actual performance.

The kind of model-based reasoning Birnbaum et al. propose was applied previously to self-debugging planners which used rule-chaining to construct plans (Birnbaum et al., 1990). Their approach was also implemented in CASTLE, a system which introspectively learned strategies for playing chess (Collins et al., 1993; Krulwich et al., 1992). A variation was applied to Firby's reactive planning domain (RAPTER, (Freed & Collins, 1994a), see below). Their proposal for applying the approach to case-based reasoning, however, was never investigated.

ROBBIE's model, while based on the proposal of Birnbaum et al., deviates from their proposed model in a number of ways. The general concept of a model of expectations used both for failure detection and explanation of failures is incorporated into our framework. ROBBIE's model, however, is hierarchical and includes highly specific information about the underlying reasoning system. Justification structures are the key element in the proposed model of Birnbaum et al. to connect the specific performance expectations to the introspective knowledge of the system. In ROBBIE, however, much of the role of justification structures is subsumed by the specific assertions incorporated in the introspective model itself, and the causal connections in the model between related assertions. Such justification structures as ROBBIE includes are used to describe a reasoning trace from which the introspective reasoner extracts information about the actual reasoning of the planner.

2.4.2 RAPTER

The RAPTER system developed by Freed applies an introspective model of ideal reasoning behavior to a Firby-like reactive planner in Firby's deliver-truck domain, Truckworld (Freed & Collins, 1994b). The model is similar to ROBBIE's and the

CASTLE model mentioned above, in that it is a set of assertions about the expected reasoning behavior of the system. Freed's approach, like that proposed in Birnbaum et al. (1991), stresses the importance of justification structures for tying together performance expectations and knowledge about the reasoning process. Once an expectation about performance is known to be violated, the justification structure associated with it is used to determine possible failed assumptions on which the failed expectation depends. In addition to introspective knowledge about its reasoning process, RAPTER's justification structures may relate a performance expectation to knowledge of the domain or general knowledge about planning. These difference classes of knowledge are used together to determine a cause and repair for a given failure. ROBBIE, by contrast, uses only its model, which contains detailed knowledge of the reasoning process and causal relationships within it, to explain detected failures. ROBBIE's hierarchy of abstract and specific assertions takes the place of RAPTER's justification structures. ROBBIE successfully performs diagnosis and repair of reasoning failures without incorporating domain knowledge or extensive justification structures.

RAPTER does not perform continuous monitoring of its planner's reasoning process, but incorporates monitoring of the *outcomes* of reasoning into the checks the planner itself makes of its sensor input and state of the world. ROBBIE does not depend on actions of its planner to determine when reasoning problems are detected; it interrupts its planner whenever possible to examine the current reasoning process. ROBBIE monitors not only the outcomes of the planning process, but the reasoning process itself to discover failures.

2.4.3 Autognostic

Autognostic incorporates a different form of model for representing reasoning processes (Stroulia & Goel, 1994). Instead of a model that contains explicit expectations about the underlying reasoning process, Autognostic represents the underlying reasoning process as a computational process. It uses an existing framework for representing the function of devices to describe the functioning of the reasoning system to which it is attached. The framework used is a "Structure-Behavior-Function" (SBF) model (Chandrasekaran, 1994; Goel & Chandrasekaran, 1989) which describes the system in terms of *tasks*, which may be thought of as goals of the system, *methods*, which implement those tasks and which may in turn contain *subtasks*, which must be achieved to apply the methods. Each task and method is defined in terms of its *structure*, the subtasks, methods, and primitive structures that make it up, its *behavior*, the process by which it achieves its goal, and its *function*, i.e. a description of what it is supposed to accomplish. From this sort of model it is possible to classify the different kinds of changes that might be made to the system, and how an alteration at oint point triggers the need for alterations in other portions of the system. It is also possible to trace the behavior of the system through the expected selection of tasks, application of methods, and knowledge transferred from one task or method to another. The model, unlike ROBBIE's, does not include explicit expectations about the performance of the system, nor does it indicate how to map underlying reasoning behavior onto the model. The focus of Stroulia's work, therefore, is on the process of assigning blame for a detected failure, not on the process of detecting the failure in the first place. To strongly support failure detection, we needed to develop a representation of desired performance.

Autognostic has been used to model the performance of two different underlying systems: ROUTER (Goel, Ali, & de Silva Garza, 1994) which applies both case-based planning and rule-based planning to construct route plans for traversing a college campus, and Kritik2 (Stroulia & Goel, 1992) which performs device design and diagnosis.

The SBF model includes more detailed kinds of information about the reasoning process than ROBBIE's model, because SBF models are designed to assist in the *design* of the modeled system as well as to diagnose failures of such systems. Our approach requires less knowledge, but is successful at an additional task: detecting reasoning failures.

2.4.4 Meta-AQUA

Several approaches to introspective reasoning have taken a different tack than the ones discussed so far, which all incorporate an explicit model of reasoning separate from the main task's reasoning process. Meta-AQUA performs failure-driven introspective learning in a case-based manner, by retrieving and applying cases which describe reasoning events instead of domain events (Ram & Cox, 1994). Meta-AQUA is a story understanding system that applies explanation pattern cases (XPs) to explain anomalies (expectation failures) discovered in the story it is given. When Meta-AQUA fails to understand a story fragment it expects to understand, it applies "Introspective Meta-XPs" (IMXPs) to correct its reasoning process. An IMXP describes the reasoning failure with which it is associated, and describes possible learning strategies for determining the exact cause of the failure and for repairing it. An Introspective Meta-XP serves as a template which may be matched against a description of the actual reasoning process (stored in "Trace Meta-XP"), to determine if the reasoning failure associated with a given IMXP has occurred. Different IMXPs refer to different classes of reasoning failures, such as "Novel situation," "Incorrect background knowledge," or "Impasse during memory retrieval," (Cox, 1994). Cox (1995) describes a taxonomy of different reasoning events and failures from which IMXPs might be derived.

Meta-AQUA focuses on the use of Meta-XPs to explain and learn from reasoning failures. It applies its meta-reasoning only when its main task, story understanding, exhibits an explicit failure itself (as when it fails to explain an entire story, or recognizes a failure to explain that was later resolved). Unlike ROBBIE, it cannot detect suboptimal reasoning performance unless later information reveals the sub-optimality explicitly.

2.4.5 IULIAN

IULIAN is another system which integrates introspective reasoning with its overall case-based task. IULIAN's task is discovery learning, to describe and explain problems presented it by forming theories or by performing experiments (Oehlmann, Edwards, & Sleeman, 1995). Its case memory contains previously solved problems, plans for creating experiments by adapting existing ones, and plans for forming theories through posing questions and recursively applying its methods to answer them. Cases may refer to IULIAN's domain task, or to introspective tasks such as monitoring the application of a reasoning strategies or explaining reasoning failures through the same methods as applied to the domain task.

IULIAN is driven by expectation failures, as are many introspective reasoners, including ROBBIE. An expectation failure for IULIAN occurs when an expected outcome due to a hypothesis or theory it has formed is shown to be false by experimentation. IULIAN retrieves an introspective plan for determining the cause of an expectation failure and recovering from it.

Improvements in IULIAN's reasoning process come about as a result of applying introspective plans that describe how to evaluate its reasoning and how to correct failures of it. Unlike ROBBIE, IULIAN does not maintain an explicit model of what its reasoning process should be: its introspective knowledge is encapsulated in its introspective plans. While ROBBIE's introspective framework is generally applicable to many different reasoning systems, it is unclear how IULIAN's approach would apply to any but a discovery learning system.

2.4.6 Massive Memory Architecture

Arcos & Plaza (1993) approach introspective reasoning using unified architecture for describing case-based and meta-level problem solving tasks by describing each process by decomposition into tasks and sub-tasks. If a domain task, such as "find the age of Mary", has no known method for solving it, then a meta-task is initiated to find such a method. The architecture supports reflective processing by representing the domain-level tasks and introspective tasks with the same framework. Failures are driven by impasses when reasoning cannot continue normally, such as just described. By contrast, ROBBIE can detect opportunities for learning which do not create impasses, and can distinguish ideal reasoning behavior from merely adequate reasoning behavior.

2.4.7 Other methods for reasoning improvement

Systems such as SOAR improve their reasoning methods by chunking the rules it uses to control the rule selection and focus of attention processes (Rosenbloom et al., 1993b, 1993a), rather than by analyzing the overall reasoning performance based on explicit representations of its desired function. In addition, the scope of circumstances from which SOAR can learn does not include all of its reasoning processes: the mechanisms for finding and applying its rules are outside of its control. In contrast, refining these processes is precisely the heart of ROBBIE's learning.

Other systems depend on interactions with a human user to detect flaws in their reasoning processes and to improve those processes through human guidance. The ASK system learns "strategic knowledge" about how to apply its domain knowledge through a dialogue with a human user who corrects mis-applications of knowledge (Gruber, 1989). The knowledge about how the reasoning process should perform is all provided by a human user: in ROBBIE that knowledge is represented in the introspective reasoner's own model, and the system itself determines the cause of a failure and how to repair it.

2.5 ROBBIE's features

In this chapter we have described the various pools of research out of which ROBBIE has sprung. ROBBIE incorporates a relatively standard case-based planner (with a few twists), but combines the case-based planner with a reactive planner based on Firby's RAP framework, to perform route planning. Of more interest than the planner is ROBBIE's approach to introspective reasoning, which relates to work done by Birnbaum, Collins, Krulwich, and Freed on using models of ideal reasoning to detect and repair failures of reasoning.

ROBBIE differs from other case-based planners by incorporating execution into the planning process. Its model of introspective reasoning is hierarchical, unlike the models to which it is most closely related, and incorporates detailed knowledge about the underlying reasoning system and the relationships between expectations, which in other models were maintained separately. It stresses the importance of both detection of expectation failures and the explanation of failures, and utilizes a single model separate from the underlying reasoner itself, to implement both tasks.

Chapter 3

ROBBIE and Its Domain

> For developing a model of introspective learning, the task and domain must be sufficiently rich for the system to potentially benefit from introspective learning without overwhelming the learning abilities of the system. In this chapter we describe the testbed domain in detail and sketch ROBBIE itself.

We are investigating introspective reasoning as applied to reasoning for a planning task, the task of creating and carrying out plans to navigate as a pedestrian from one location to another. The task has several parts: first, *route planning* in which a plan is made deliberatively which should get from the starting location to the goal, second, actual *execution* of the plan which was created, and last, *learning* from the success or failure of the original plan. All three parts of the task are important. Without some sort of planning ahead the pedestrian would have no guidance in finding its way to the goal location, without executing the plan it has no way to verify that its planning was correct, and without learning from the results, it would repeat mistakes made in planning or executing a plan, or waste effort re-creating a route already experienced, over and over.

Let us consider one example of the kind of problem ROBBIE faces, involving the world map shown in Figure 3.1. A person interacting with ROBBIE selects a starting location, such as the middle of the block of Birch street between Elm and Oak, and then specifies a goal location, like the corner of Fir and Oak streets. From this description of starting and goal locations, ROBBIE uses case-based reasoning to generate a plan for moving from start to the goal, described in high-level plan steps like:

- Turn east on Birch
- Move east to Oak
- Turn north on Oak
- Move north to Fir and Oak.

ROBBIE then executes this plan in the simulated world of which it is a part. If the plan succeeds without a hitch, ROBBIE should learn by remembering this route as a success. If ROBBIE encounters some problems in executing the plan, for instance if Oak is closed beyond Apple because of long-term construction, then ROBBIE might find another way to the goal (perhaps by detouring on Apple and Cedar). ROBBIE should learn that the route it selected was flawed, and to pay more attention to where construction is in the future. If the plan itself were incorrect, if it had "south" instead of "north" in the second pair of steps, then ROBBIE might become completely lost and fail to reach its goal. It still has the opportunity to learn from this situation by examining how its reasoning went awry, and improving its plan creation process.

The description of the domain at the beginning of this chapter does not give a clear sense for how detailed or complex the knowledge available to ROBBIE's reasoning system actually is. We could be describing a problem as simple in form and

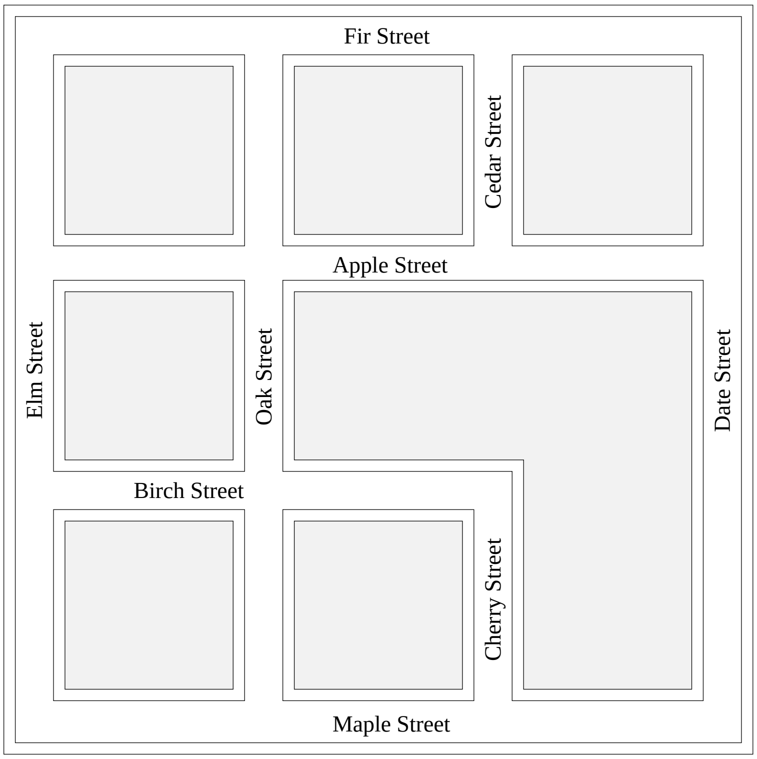

Figure 3.1: One of ROBBIE's worlds

knowledge as the standard traveling salesperson problem, or as complicated as a real physical robot moving about on real world streets. In setting up the actual domain implementation, we must determine a point which is both feasible and interesting for our task between these two extremes on the spectrum of complexity. A good implementation of the domain for our purposes provides a level of complexity which is manageable while being complex enough to lead to opportunities for introspective learning. The description of locations and objects in the world must be sufficiently knowledge-rich to make it difficult to correctly weight the relative importance of the features a priori. It should be difficult to determine which feature are important to include, and which interactions between features are important. Both features and interactions may be implicitly part of the input description without being explicitly enumerated (for instance, the concept of two locations being "on the same street" as a feature connecting starting and ending locations is not explicit in ROBBIE's input problem description). Beyond the richness of the representation of objects in the world, the domain should also include dynamic elements. Elements which change out of the control of ROBBIE ensure that static knowledge is insufficient to capture the state of the domain, and provide more opportunities to learn from experience. Most importantly, the complexity of the domain description should be extensible beyond the original description to more knowledge-rich and more dynamic aspects as time and ROBBIE's sophistication permit.

In this chapter we will first describe the domain in which ROBBIE plans and learns in more detail, and we will discuss how our choices in creating this domain address the issues raised in the previous paragraph. We will then provide an overview of the ROBBIE system and the world simulator which interacts with it. We will describe how the world simulator implements the domain for the planning task, and will briefly describe the major components of the ROBBIE system itself. We will focus in this chapter on how control and information passes between ROBBIE and the world simulator and from component to component within ROBBIE itself, and will provide details of ROBBIE's implementation in later chapters.

3.1 The domain

The domain chosen to test out the introspective reasoner is that of a planner for a robot moving from place to place on a simple grid of streets as a pedestrian. Figure 3.1 above shows one of several typical world maps in which ROBBIE has been tested. The simulated robot starts out at some given location on a sidewalk of the world, and ROBBIE is given another sidewalk location as a goal. ROBBIE creates and executes

- Turn north on the corner of Maple and Elm
- Move north on Elm to the north side of Birch
- Turn east on the north side of Birch
- Move east on Birch to the position 20 units along Birch

Figure 3.2: High-level plan

a plan to move the robot from its current location to the goal location. Plans created by ROBBIE describe how to reach the goal location with high-level plan steps, like the one shown in Figure 3.2. These high-level plan steps are treated as sequential goals of the reactive execution component of the planner.

Locations in this domain are not simple points on a graph, nor pairs of Cartesian coordinates, but descriptions in terms of street names, sides of streets, blocks, and so forth. These detailed descriptions provide many different levels of comparison between locations, and permit intuitive descriptions like "the north side of Birch" which vary in specificity from describing a whole side of a street to a particular point along a street. The domain also includes dynamic elements which the planner is expected to handle as a matter of course: traffic lights (ROBBIE is never to cross against the light), blocked streets, broken traffic lights, and so forth. The planner is also expected to recover, if possible, from failures of its plans.

This domain is a good one for testing the limits of ROBBIE's introspective learning abilities because of the complexity of information available to the system, the dynamic elements included in the domain, and its extensibility. Locations are complex entities containing a great deal of information to be integrated, along with additional information such as street signs, traffic signals, and nearby obstacles. Moving about successfully requires coordinating the available information and anticipating and reacting to many variables out of the planner's control, such as traffic lights and closed streets. In a domain of this type it is difficult to predict all the ways of responding to new situations that the planning system will need; hence the need for introspective reasoning is greater. New responses may be added as needed in a system such as ROBBIE where reasoning itself may be learned. The extensibility of the domain makes working with it tractable while allowing it to scale up: at the current time many possible complications and dynamic elements have been omitted, but can be included at a later time to test ROBBIE's robustness under increasing complexity (for instance, other pedestrians, cars, time constraints, etc.).

3.2 Overview of the system

Figure 3.3 shows a view of the entire system, consisting of ROBBIE itself and the simulator that controls the world in which ROBBIE is situated. The simulator maintains the state of the objects in the world and updates their state at each time step. It informs ROBBIE of the state of the world through a set of "sensory" values provided to ROBBIE at each time step which, aside from goal locations received directly from a human user, are ROBBIE's only inputs. In return the world simulator receives from ROBBIE instructions for low-level actions for the simulated robot to take. The simulator interprets those actions and updates the robot's location and state in response to them.[1]

ROBBIE itself contains two components, the planner which interacts with the

[1]We will use "ROBBIE" only to refer to the reasoning system which creates plans and performs introspective reasoning on itself, and we will use "the simulated robot" to refer to the object in the simulated world which represents ROBBIE.

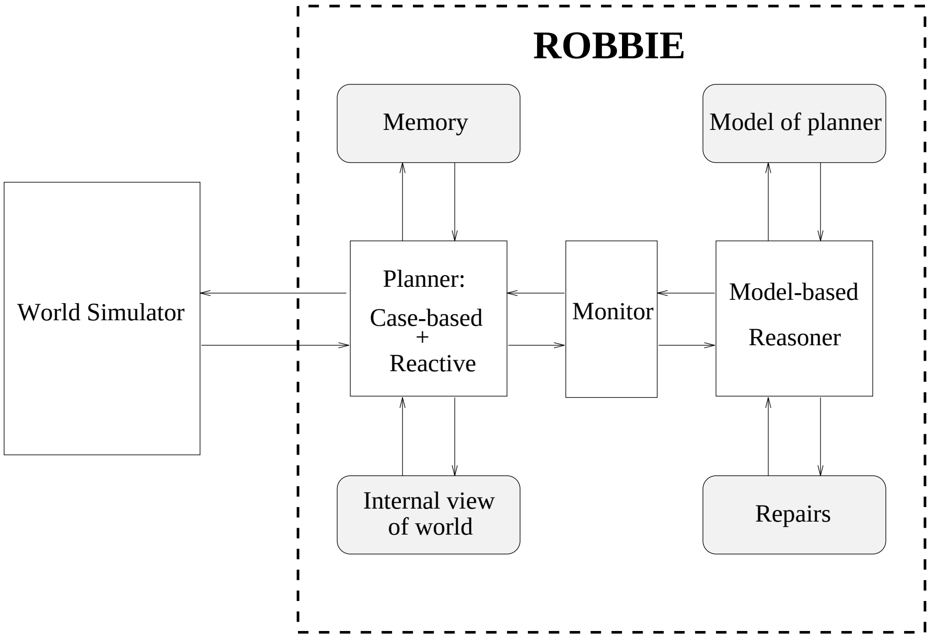

Figure 3.3: ROBBIE and its world simulator

simulated world and handles the *performance task* of creating and executing plans, and the introspective reasoner, which watches the reasoning behavior of the planner and corrects it in response to reasoning failures. The introspective reasoner has no direct contact with the world simulator, and communicates with the planner through a special, limited protocol which determines when and where introspective interference is allowed and which permits the planner to be informed of the results of introspective reasoning. In the following sections the world simulator will be described in detail as an implementation of the domain, and the components of ROBBIE will be sketched to explain their relationship to each other.

3.3 The simulated world

The world simulator implements the domain described in Section 3.1, and regulates the relationship between ROBBIE's actions and the domain. It keeps track of time,

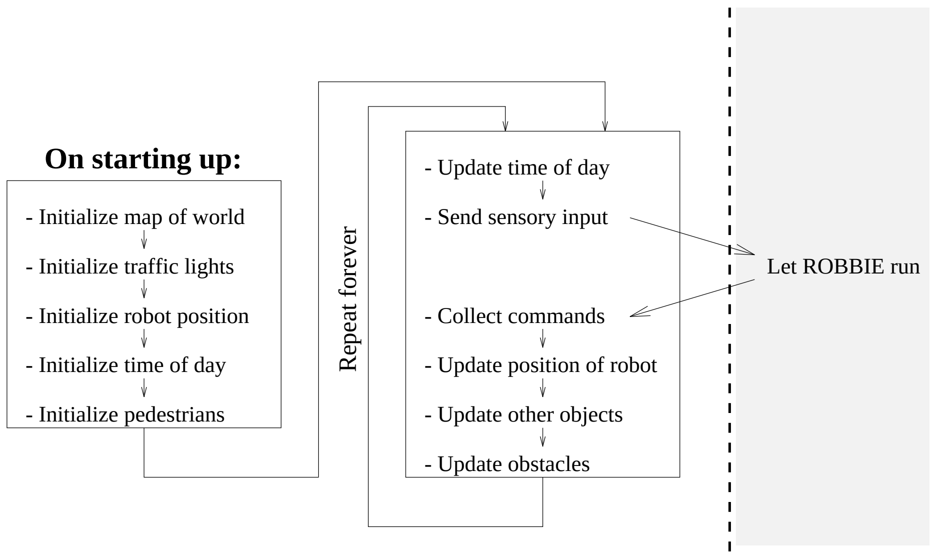

Figure 3.4: Simulator's process

updates the state of objects in the world at each time step, generates sensory input for ROBBIE and updates ROBBIE's state and position based on output command ROBBIE sends it. Figure 3.4 shows an overview of the world simulator's process.

The streets on which ROBBIE travels are laid out in a grid pattern, although not every street runs the whole length or width of the map (as in Figure 3.1). ROBBIE's simulated robot is a pedestrian, and is therefore expected to travel on the sidewalks and cross streets only when the traffic light is green, but that restriction is not incorporated in the world simulator, which will permit the robot to move to any point on a sidewalk or in a street, and only restricts the robot from moving into any "buildings" (the space of each block inside the sidewalks).

The simulator maintains a map of the world which consists of discretely measurable objects: blocks are 100 units long, sidewalks are 6 units wide, and streets have varying width, with a default of 20 units. Blocks are numbered sequentially starting in the southwest corner and increasing to the east and the north (i.e., the west-most block is numbered with zero, the next eastward block is 100, and so on).

```
(sidewalk
   (street maple)
   (side north)
   (block 100)
   (along 20)
   (across 3))
```

Figure 3.5: A typical `sidewalk` location description

Any point on a sidewalk or in a street has a *location description* which the world simulator can use to determine the simulated robot's location, or can pass to ROBBIE to describe a location of interest in the world. While in theory any describable location could be given to ROBBIE as a goal, in practice goals are restricted to locations which are on sidewalks in keeping with the pedestrian nature of the robot. Location descriptions are *frames* which are classified by the type of location each describes; the slots for each class of frames differ depending on the features of that type of location.

The canonical location type is the `sidewalk` form which is used whenever the simulated robot is standing on a sidewalk in the middle of a block, excluding sidewalks at the intersection of two streets and places where one block changes into another. Figure 3.5 shows a typical sidewalk location description. The information in a sidewalk description includes the `street` beside which the sidewalk runs, the `side` of the street on which the sidewalk sits, the `block` of the street, the distance `along` that block on the sidewalk (a number between 1 and 100 since blocks are 100 units long), and the distance `across` the sidewalk (a number between 1 and 6). The `across` value is often ignored by ROBBIE itself, but is important to the world simulator in order to maintain the exact location of the simulated robot.

Other location types include `intersect` for locations on the sidewalk at the intersection of two streets, `in-street-inters` for locations in the street at an intersection (as when the robot is crossing a street), `street` for locations in the street *outside* of

```
(intersect
   (street1 oak)
   (side1 east)
   (block1 300)
   (along1 97)
   (street2 fir)
   (side2 south)
   (block2 200)
   (along2 4))
```

```
(in-street-inters
   (street1 oak)
   (side1 west)
   (block1 (0 100))
   (along1 12)
   (street2 maple)
   (side2 in)
   (block2 100)
   (along2 97)
   (width2 20))
```

Figure 3.6: Typical `intersect` and `in-street-inters` location descriptions

```
(street
   (street elm)
   (block 100)
   (along 75)
   (across 5)
   (width 20))
```

```
(between-blocks
   (street date)
   (side west)
   (block (100 200))
   (along 18)
   (across 3))
```

Figure 3.7: Typical `street` and `between-blocks` location descriptions

an intersection, and `between-blocks` for sidewalks where a street normally would be. Figure 3.6 and Figure 3.7 show examples of each of these location types, and Figure 3.8 shows a portion of the map in Figure 3.1 with the different location types indicated by shaded sections.[2] The intersection location types describe the robot's location in terms of both streets at which it is standing; the first street in the description is the street along which the robot is facing.

By using a more knowledge-rich representation of locations than simple Cartesian coordinates, we provide ROBBIE with a better view of its world than it would otherwise have. Locations are described in a way that is more intuitive for people to understand, and captures information that can make ROBBIE's planning task easier (for example, `intersect` locations are important for the ability to change direction,

[2]ROBBIE currently treats all intersections as four-directional, and sidewalks at an intersection where a street would be as equivalent to being in the street itself.

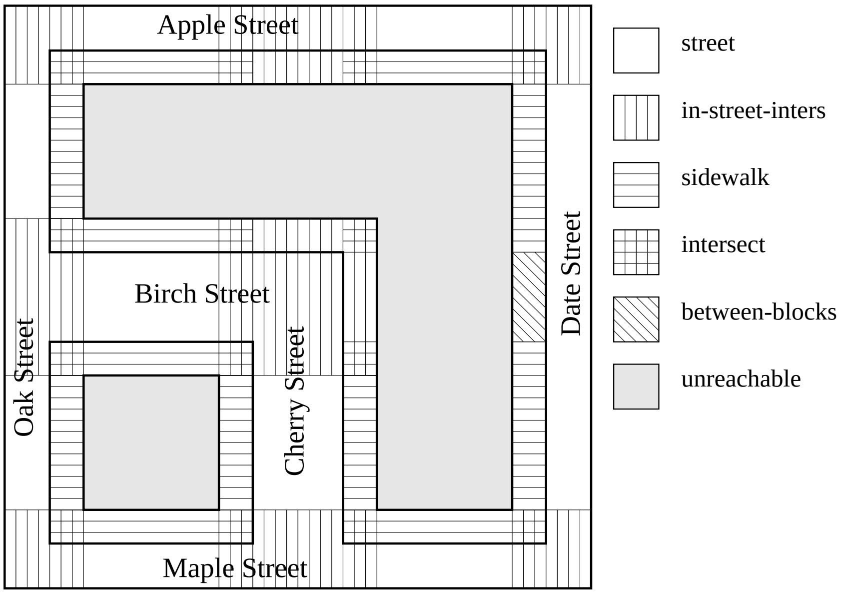

Figure 3.8: A portion of the world map with location types marked in shaded sections

and can be easily distinguished from `sidewalk` locations with this method). It is also easy to represent inexact locations like "the north side of Fir street" or "the 100 block of Oak" simply by leaving some slots of the location frame empty.

The world simulator updates and maintains the dynamic elements of the domain. The most important are the traffic lights at intersections which control when ROBBIE permits a street crossing. Currently, all intersections have traffic lights which operate independently of each other and of ROBBIE. Each light changes from red to green for a particular direction, on a 30 time step cycle. When the world simulator is initialized, each traffic light is set to a random point in its 30 time step cycle. Traffic lights sometimes break down, remaining green in one direction for a random amount of time. Other dynamic elements are implemented in the world simulator but are currently ignored by ROBBIE: they exist for future expansion of the complexity. Streets may be unexpectedly blocked for a random amount of time, and other pedestrians are located on the map in differing numbers depending on the calculated time of day (many pedestrians at rush hour, few at 3 o'clock in the morning). At this time, the world simulator allows ROBBIE to walk through closed streets and other pedestrians as if they were not there, but their existence in the simulator lays the groundwork for further extensions.

The world simulator provides input about the simulated world to ROBBIE, informing it of the simulated robot's internal state and about objects and events in the world which are within some distance of the simulated robot and to which the robot is paying attention. The internal state consists of the direction the robot is facing, whether it is standing still or moving, how fast it is moving, and how far it moved in the last time step. ROBBIE is not given the new location of the simulated robot, but must determine it from moment-to-moment information and knowledge of its original position. ROBBIE can move at two speeds: "slow" is one unit per time step, "fast" is three units per time step. External objects whose presence is reported by default include walls, streets, and intersections within 20 units of the robot (in any direction). ROBBIE is also informed when it is changing from one location type to another (i.e., when it moves from an intersection into the street). ROBBIE may request information about other objects in the world, including the status of traffic signals and street signs. After ROBBIE requests such information, the presence and status will be reported at each time step until ROBBIE commands the simulator to cease reporting that object.

ROBBIE must send commands to the simulator to control the simulated robot, including control over which "attentional" inputs will be reported. Table 1 lists the possible commands ROBBIE can use. These simple commands are combined by ROBBIE to produce more complex behavior.

From this discussion it should be clear that the world simulator operates almost

`stop`	Stop the robot from moving
`move`	Start the robot moving with the current direction and speed
`slow`	Set the movement speed to slow
`speed-up`	Set the movement speed to fast
`turn`	Turn the robot to a new direction
`look-at`	Report the status of the given object
`look-away`	Stop reporting the status of the given object

Table 1: Commands from ROBBIE to Simulator

independently of ROBBIE, and has control over information about the simulated world that ROBBIE needs in order to create and carry out its plans. The two systems interact only by "sensory" information provided by the simulator to ROBBIE and by commands sent to the simulator by ROBBIE to control the state of the simulated robot. This implementation of a navigational domain meets the criteria we described earlier by incorporating considerable complexity both in the richness of the available information and the dynamic nature of the domain. The complexity is restricted to a manageable level, while at the same time allowing for significant future expansion. The simulator already includes some elements of the domain which are, for expediency, ignored by the robot and ROBBIE, but which could easily be activated. Additional objects in the world such as cars and features such as time constraints on ROBBIE's performance could also be added using the same basic simulator mechanisms.

3.4 The planner

ROBBIE's planner combines a typical case-based planner (Hammond, 1989) with a reactive planning execution module (Firby, 1989). Figure 3.9 outlines the planner's process. The case-based planner takes a description of its current situation (the current starting and goal locations), builds an index describing the situation, and

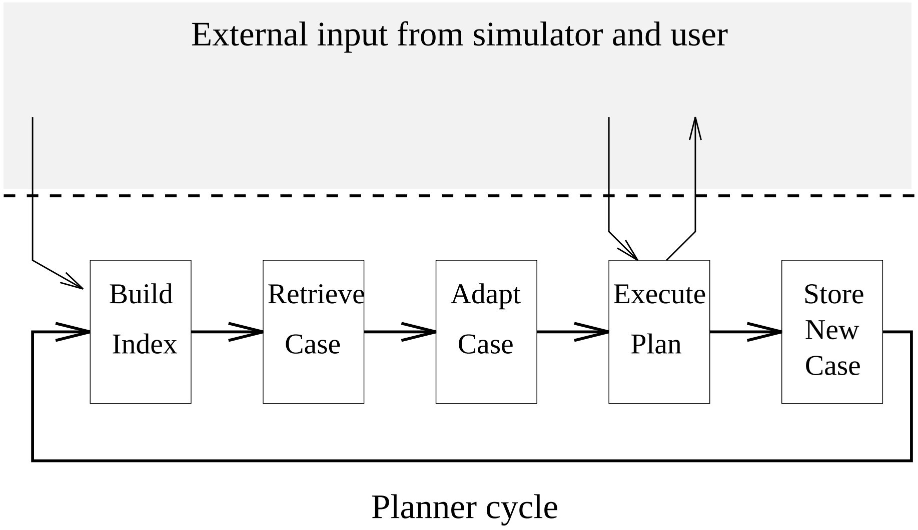

Figure 3.9: ROBBIE's planning component

retrieves old plans with similar indices for already-known routes on the current map. Initially similarity is judged by the combining the geographical proximity of the current starting location to the starting location of the old case, and the geographical proximity of the current goal to the goal of the old case. ROBBIE learns new criteria for judging similarity through introspective reasoning: the new criteria describe features which were implicit in the original description of a situation but were not made explicit in the index. ROBBIE applies a set of adaptation rules to convert the most similar old case into a route plan which is applicable to the current situation. Each rule makes a small local change which builds from the old case toward a new case.

Because retrieval and adaptation are guided by vague concepts like similarity and heuristic rules, evaluation of a CBR-created plan is very important. ROBBIE performs evaluation by reactively executing the adapted plan to see how it plays out in the simulated world. This execution also forms a last stage of adaptation, as the reactive execution may eliminate useless steps or add new ones in response to obstacles.

The reactive planning component is based on Firby's RAP model. Each high-level plan step created by the case-based planner is considered in turn as a goal for the reactive planner to achieve. At each time step, the reactive planner considers the current input from the world and its current goal and selects a new reactive *planlet* to execute. Each planlet contains commands to the planner itself (such as recording some piece of information) and commands to the world simulator (such as speeding up or slowing down).

After the goal location has been reached through execution of the plan, the final plan is reconstructed from the actions of the reactive planner (most often it is the same as the adapted plan). This plan is stored along with the index (the description of the situation which was used to retrieve a case before), and the planner requests a new goal location. By default, the new situation will have the current location of the robot as its starting point, and awaits only a new goal location from the user.

3.5 The introspective reasoner

The introspective reasoner is signaled to examine its expectations by the planner when the planner is in a state in which introspective monitoring can occur. This state could be a point in the planner's reasoning process which represents a completed task (for instance, when retrieval has finished considering each case in memory). It could also be a point at which the planner has noticed a catastrophic failure and should not continue without introspective help (for instance, if the execution component has become lost). A description of the current situation in a restricted monitoring vocabulary is passed to the introspective reasoner.

Figure 3.10 shows the general structure of the introspective reasoning process.

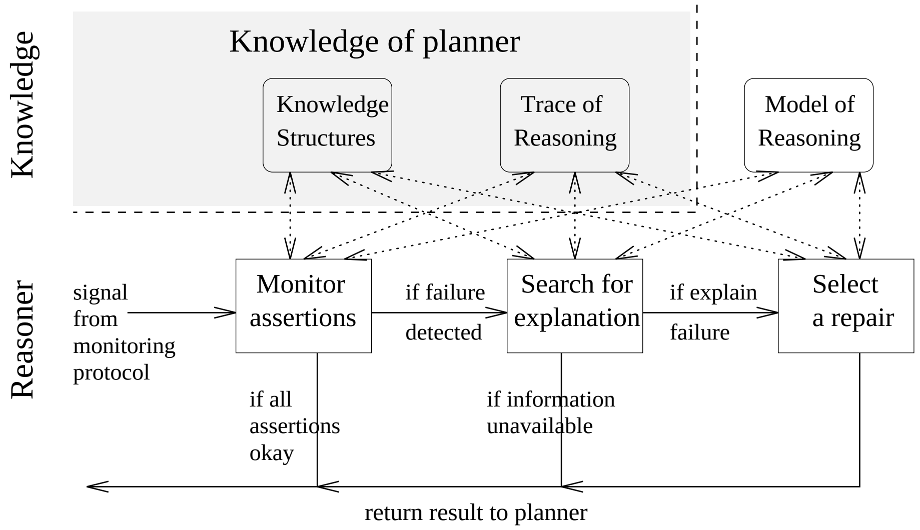

Figure 3.10: Structure of the introspective reasoner

The introspective reasoner examines the current situation for expectation failures, drawing on its model of the planning process, the trace of reasoning so far, and any knowledge structures the planner may have created. An expectation failure occurs when one of the model's assertions about the current ideal state of the reasoning process proves to be false of the actual reasoning process. This is generally assumed to be true if the planner encounters a catastrophic failure and cannot continue.

If the introspective reasoner finds no expectation failures, meaning that all assertions about what *should* be happening are true of the actual reasoning process, then it returns control to the planner, informing the planner that everything is okay. If a failure is discovered, however, the introspective reasoning process moves on to search for the original cause of the failure by explaining how the failure came about. Once again it may draw on the knowledge structures and reasoning trace of the planner as well as its own model of the ideal planning process.

Because the introspective reasoner may need to access data structures of the planner, it is possible that at the time a failure is detected, the information available is not sufficient to diagnose and repair the failure. Therefore, the introspective reasoner may decide to suspend its explanation and repair tasks and permit the planner to continue until more information is available. When the requisite information becomes available, the explanation and repair task will be restarted from where it left off. If the planner is unable to continue in any way, then the system must simply fail to achieve its current goal and fail to explain the failure, a situation we hope will be rare.

The last module of the introspective task involves creating a repair to prevent the detected problem in the future. The repair will alter the reasoning related to the expectation failure that was determined to be the root cause of the originally detected expectation failure. The explanation of the original failure provides clues to help the introspective reasoner repair the failure, as may the actual root cause expectation. From this information the repair module creates a repair and alters the underlying planning process. In the case of ROBBIE, repairs may be applied only for indexing problems, which form the bulk of failures for ROBBIE in any case. An indexing repair requires discovering a new feature for indexing cases in memory, adding a rule to look for that feature in future problems, and re-indexing the cases in memory to include the new feature. Once the repair has been accomplished, or the impossibility of a repair is determined, the planner is given control again and informed of what took place.

3.6 The ROBBIE system

In this chapter we have explained the details of ROBBIE's domain and the reasons for our choices in designing it. In addition, we have given an overview of the system as a whole and a brief summary of how each part works.

The domain is one which should be difficult for an ordinary deliberative planner to handle because of the richness of information about any given situation, the imperfect knowledge available to the planner, and the presence of dynamic elements which will not remain the same throughout the planning or execution processes. ROBBIE has only incomplete initial knowledge of the domain, and the features of the domain are rich enough to make determining feature relevance for a given situation difficult. Because of this, ROBBIE's domain should provide many opportunities for introspective learning to fit ROBBIE's reasoning mechanisms to the requirements of the domain. Some of the dynamic elements of the domain are addressed by the planner itself, since it is a combination of case-based planning with reactive execution planning. The feature selection problem is addressed, however, through introspective learning which looks for and includes new features from a given situation when the existing features prove to be inadequate.

In the next chapters we will look at the components of ROBBIE in detail, examining how each works and what issues were addressed in implementing each. We will first describe the planner and provide sample output from ROBBIE's planning process. In following chapters we will discuss "model-based reasoning" as the foundation for introspective reasoning in general, and then our specific implementation of those ideas.

Chapter 4

ROBBIE's Planner

ROBBIE's planning component combines a case-based, deliberative planner with a reactive planner for execution. Subparts of the planner use the same case-based process as the planner as a whole. In this chapter we describe the planner and discuss the issues that it addresses.

The ROBBIE project investigates how introspective learning can improve the performance of a system for a domain task: in ROBBIE, that domain task is navigation by a robot pedestrian. This task is guided by a planner that takes a goal, which is a desired new location for the robot with an attendant starting location, and plans a route to get to that goal. ROBBIE interacts with its simulated world to execute the new plan and test it "under fire." These two steps require a combination of deliberative planning[1] using case-based reasoning and reactive execution which responds to the problems that inevitably arise in applying an abstract plan to a concrete situation. Thus the components of ROBBIE's planning process are: goal generation, retrieval, adaptation of retrieved cases, plan execution, and case storage. To introduce these components, in this section we highlight each in turn. In the remainder of the chapter

[1]Recall that deliberative planners plan out the entire problem before executing it.

we examine in depth the issues they involve.

Goal generation: Goals for ROBBIE are simply pairs of locations: where to start from and where to end up. Most of the time, goals arise from external sources such as a human user: the human plays the role of a dispatcher, as ROBBIE plays the role of a pedestrian delivery robot.

ROBBIE's goals can arise in three ways. First, the user may provide a single location, with the implicit command for ROBBIE to plan *and execute* a route from its current location to the new location. Second, the user may specify a sequence of locations, to be achieved one after the other like goals in a treasure hunt. Since ROBBIE considers each goal location in the list one at a time, this type of input goal is essentially the same from ROBBIE's perspective as providing goal locations one at a time. The third kind of goal comes not from a human user, but arises from ROBBIE internally through the process of "re-retrieval," which we will discuss below. In re-retrieval, ROBBIE splits the given problem into two subparts by selecting an intermediate location, and then generates two subgoals for itself, to create (but not execute) a plan from its current location to the intermediate, and to create (but not execute) a plan from the intermediate to the goal. These sub-plans are then combined and executed as a single plan.

Case retrieval: Once a goal has been created, ROBBIE must generate a route plan that describes a path from its starting location to the goal location. The plan is created by a case-based planner structurally similar to the planning system CHEF (Hammond, 1989)[2]. The case-based planner takes the goal (the new desired location and current situation) and builds an *index* which describes the situation in terms of

[2]Previous case-based route planners are discussed in Chapter 2.

its relevant features and how cases are stored in memory. The index is compared to the indices of cases in memory to select the one with the most similar index. This case forms the basis of a solution to the new problem.

Case adaptation: The retrieved case is adapted to create a case which applies to the current situation exactly. Adaptation in ROBBIE involves mapping the locations of the retrieved case onto the locations of the current problem and then filling in the intervening steps. The Adaptor applies many individual adaptation strategies which alter small portions of the information in the old case to create a new solution.

ROBBIE's adaptation mechanism includes one aspect beyond a straightforward case-based approach. Traditional CBR systems may be stymied if a retrieved case is unadaptable. A case may prove unadaptable either because a faulty retrieval chose an inappropriate case when a correct alternative existed, because all cases in memory are dissimilar to the current situation so that no match is close enough to be mapped, or because the current situation requires a more complex plan than any in memory. When ROBBIE retrieves a case that cannot be mapped onto the current situation, regardless of the cause, ROBBIE uses its *re-retrieval* process to alter the planning goal of the system. The original goal is broken into more easily solvable pieces. A location lying between the starting and goal locations of the original problem is chosen and normal retrieval and adaptation are used to create two partial solutions, (one from the starting location to the intermediate location, one from the intermediate to the goal). The two new solutions are merged to form a solution to the entire problem.

Plan execution: Execution of the new adapted plan in the simulated environment serves to evaluate the quality of the CBR-created plan, and it also acts as a last component of adaptation, as the reactive Executor may alter the plan steps as needed to solve the problem.

Reactive execution is based on Firby's RAP approach to reactive planning. The plan steps created by the case-based planner are treated as goals to be achieved in sequence by the Executor. A step like "Move east to Oak" becomes a goal meaning "Apply planlets until the simulated robot reaches Oak." At each time step, the reactive Executor selects an appropriate planlet to help achieve the current reactive goal. Between one time step and another the context may change drastically, requiring a completely different response. Reactive execution alters its behavior in response to momentary changes in its environment. Each action of the reactive system is recorded, allowing reconstruction of the actual steps taken after the goal location has been reached.

Plan storage: The final solution to a given problem is the result of both deliberative planning which creates the framework of a solution, and reactive planning which alters the plan to fit the actual world. The solution is reconstructed from the steps the reactive execution component took in reaching the goal. The reconstructed plan is then stored into memory, forming the learning portion of the case-based learning process. Once in memory, the case may be retrieved and re-applied whenever similar situations arise.

In this chapter we will first discuss a central feature of the planner's design, the re-use of the case-based retrieval process for subparts of the case-based reasoning process itself. We will then examine each component of the planning process in turn, illustrating its operation with sample output, and discussing the issues to be addressed in implementing each task.

4.1 Using CBR to implement CBR

In developing ROBBIE's planner, we noticed that certain components of the case-based planning process were similar to one another, and that their function was similar to the overarching case-based reasoning process itself. We chose to explore that similarity by using the same basic CBR process for all these parts: the same case memory to hold the knowledge for each component as well as the planner, and the same indexing and retrieval mechanisms to access the knowledge. Using the same case-based mechanisms for knowledge of components as for the whole is an elegant approach, and suggests the possibility of extending the re-use to adaptation and learning of knowledge for parts of the case-based process as well as the process as a whole.

In the current system the Indexer, the Adaptor, and the Executor all have case-based retrieval at the heart of their processing. Table 2 outlines how the knowledge of each part relates to the others, and sketches how expectations about each component's process match each other. The Indexer uses rules which specify when to alter the current index; those rules are retrieved from memory using a description of the current index to retrieve applicable indexing rules. The Adaptor applies strategies for altering portions of a retrieved plan. These strategies are stored in memory and retrieved by a description of the portion involved. The Executor uses reactive planlets to execute steps of the high-level plan. Each planlet is applicable in certain situations in the world; a description of the current world situation serves as an index for retrieving them.

The Indexer and Retriever must build indices and retrieve cases for all the kinds of memory structures in the case memory. This requires general mechanisms, combined

	Memory	Expectations
Indexer	rules	Indexer will have all possible rules
		Indexer will select the right rule
		Indexer will find a rule if it exists
Adaptor	strategies	Adaptor will have all necessary strategies
		Adaptor will use the correct strategy
		Adaptor will use a strategy if it exists
Executor	planlets	Executor will have a planlet for every situation
		Executor will take the right action
		Executor will apply a planlet when it's applicable
Planner	plans	Planner will have some plan that matches
		Planner will select the best plan
		Planner will retrieve a plan if it is in memory

Table 2: Similarities between modules

with task-specific indexing schemes and similarity measures for different memory structures.

Using case-based retrieval for components of the case-based planner leads naturally to the idea of including adaptation and learning of other memory structures than plans. However, we have left the problem of adapting adaptation strategies and other such structures for the future. In ROBBIE, other memory structures are added to the case memory only through introspective reasoning, which adds new indexing rules to the system's memory.

4.2 Memory organization and cases

ROBBIE's memory contains route plans, indexing rules, adaptation strategies, and reactive planlets. Since efficient *memory access* was not a focus of this research and has been examined in detail by others, we chose the most simple method of access to implement (if not the most efficient): an unsorted list of memory structures. Our focus in memory organization was on the contents of the *indices* by which similar cases are chosen, not how cases were physically stored in memory. Once an appropriate set of indices has been selected, a number of efficient retrieval methods are available (e.g., (Kolodner, 1993a)).

A larger case memory would certainly require a more sophisticated memory access method, but ROBBIE's memory rarely grows beyond 100 to 200 cases. Another linear search is done when a new case is stored in memory, to ensure that two copies of the same case are not stored.

Each structure in memory has a name, for human convenience only, an index, and the contents of the case itself. The index describes the situation to which that structure applies and what kind of a structure it is.

In later sections, as appropriate, we will describe the memory structures associated with each component (indexing rules, adaptation strategies, and planlets), but we will describe here what a plan case contains, as plans belong to the planning process as a whole. Figure 4.1 shows a sample plan, one of ROBBIE's initial plans for the map we have used throughout. As described above, the first part of the case is the case name: `old1` indicates it is an original case, later cases are given uniquely created names.[3] Following the name is the index, which identifies the case as a plan, and then gives the starting and ending locations for the plan. If any additional features had been learned through introspective reasoning, those special indices would appear after the ending location. The body of the plan is simply a sequence of high-level plan steps which, when executed one after the other, should lead from the starting to the goal location.

Plans for ROBBIE can contain four different kinds of steps, described in Figure 4.2. The step `starting-at` makes sure the simulated robot is where it is supposed to be

[3] New case names are created using `gensym`.

```
(old1
  (plan (sidewalk (street elm) (side east) (block 100) (along 10) (across unspec))
        (sidewalk (street birch)
                  (side north)
                  (block 100)
                  (along 1)
                  (across unspec)))
  ((starting-at (dir any)
      (on (sidewalk (street elm)
                    (side east)
                    (block 100)
                    (along 10)
                    (across unspec))))
   (turn (dir north)
         (on (sidewalk (street elm) (side east) (block 100) (along 10))))
   (move (dir north)
         (on (sidewalk (street elm) (side east) (block 100)))
         (to (sidewalk (street birch) (side north) (block 100) (along 1))))
   (ending-at (dir any)
      (on (sidewalk (street birch) (side north) (block 100) (along 1))))))
```

Figure 4.1: A typical ROBBIE route plan

before taking any actions, and `ending-at` makes sure the robot is at the goal when all actions have been done. The step `turn` indicates a turn in a given direction, when the robot is at a particular location. The fourth kind of step, `move`, states the goal to move in a particular direction to a particular location, and move *on* another location. "On" locations for `move` steps usually describe whole blocks of a street, or whole sides of a street, since moving will eventually change any more detailed feature of the robot's location.

Plan cases are accessed by indices that represent aspects of their content which are available without knowing the actual route to be taken. The Indexer creates such indices, given the goal's starting and ending locations.

```
(starting-at (dir <direction>)          (ending-at (dir <direction>)
             (on <location>))                      (on <location>))

(turn (dir <direction>)                 (move (dir <direction>)
      (on  <location>))                       (on <location>)
                                              (to <location>))
```

Figure 4.2: Plan step categories

4.3 Indexer: index creation

ROBBIE must convert a goal, whether externally or internally generated, into a description of the current situation, an index, which will enable retrieval of the most similar case in memory. The most similar case should be the case that can be most effectively adapted to a good solution. Index creation involves selecting and structuring information about the current problem to match the indices of cases in memory. It may also include deriving information which is implicit in the problem description. Elaboration of the original problem description permits explicit comparison of situations on the basis of abstract features which would not be explicitly included in the input description.

As mentioned in Section 4.2, the case memory contains memory objects for the planner as a whole, and for several components of the CBR process: the Indexer, the Adaptor, and the Executor. The Indexer, therefore, builds indices for the memory structures of each of these components as well as the planner, given descriptions of the different situations for which cases must be retrieved (planning, adaptation, etc.). The Indexer itself uses case-based retrieval and, hence, recursively applies itself. This recursiveness raises interesting issues we will discuss below.

In the first sample run below, the user inputs a goal location, which is sent to the indexer:[4]

```
Rob(CBR)>> No current goal; please input next destination: ind2
Rob(CBR)>> New destination is:
The east side of elm, the 200 block, 90 feet along

Rob(Ind)>> Creating plan index
Rob(Ind)>> Creating index rule index        Recursive call: see if any
Rob(Ind)>> Index is:                        indexing rules apply to
(indexer plan 40 23 23 230)                 current problem.
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Retrieval level at 2
Rob(Ret)>> Retrieval level at 1
Rob(Ret)>> Retrieval level at 0
Rob(Ret)>> Cases under consideration are:
  ()                                        No indexing rules apply.
Rob(Ind)>> Index is:
(plan (sidewalk
        (street maple)                      Plan index is just starting
        (side north)                        and ending locations
        (block 100)
        (along 20)
        (across 3))
      (sidewalk (street elm) (side east) (block 200) (along 90)))
```

The first step in creating an index is to convert the problem description into the appropriate index frame. For plans, this process is simply a matter of putting starting and goal locations into the index structure. The final index in the sample run above is just a descriptor declaring it to be a `plan` index, and the starting and ending locations themselves. Other memory structures have different index frames labeled

[4] In all sample output, output of ROBBIE is tagged by `ROB(...)>>`, where the parentheses describe the portion of ROBBIE's process involved: `Ind` for the Indexer, `Ret` for the Retriever, `Ada` for the Adaptor, `Reret` for the Reretriever, and `Sto` for the Storer. All other output is generated by the world simulator.

by the component the object belongs to: `planlet`, `adapt`, and `indexer`. The sample run above includes an `indexer` index used to determine if any indexing rules apply to the current `plan` index being constructed. Each different index form requires different input information to the Indexer, and the Indexer creates a different structure for each form. As an example, Figure 4.3 shows the information that would be provided to the Indexer by the Executor to retrieve a reactive planlet, and the resulting index. In this case, the planlet to be retrieved involves a move where the robot is already facing the right direction and moving, and there is a street about to be entered in the same direction. Naturally, this kind of Executor request would arise much later in the planning process than the sample run above, when a high-level plan has been retrieved, adapted, and is being executed.

Often the basic index created by the Indexer fully describes the current situation for retrieval, as in the first sample run above. Sometimes, however, other features must be added to complete the problem description: specializations of the basic index. Specializations in ROBBIE are exclusively applied to `plan` indices for the main planning process, not to `indexer`, `adapt`, or `planlet` indices, and are made up of extra features which ROBBIE has learned to include in its indexing scheme through introspective learning.

To determine what features to add to a basic index, the Indexer applies relevant indexing rules which are stored in the case memory. To access them, the Indexer creates from the basic index a special `indexer` index which is used to retrieve the indexing rules from memory. In the first sample run above, an `indexer` index is created, but no indexing rules apply to the current plan situation. The `indexer` index describes the type of index to be specialized, in this case a `plan` index, and then provides other relevant information. For `plan` cases the Indexer converts location descriptions into a

Input description:

Current plan step:

```
(move
  (dir north)
  (on (sidewalk (street elm)
                (side east)
                (block 100)))
  (to (sidewalk (street birch)
                (side north)
                (block 100)
                (along 1))))
```

Sensory information:

```
((cross-street north))
```

Resulting index:

```
(planlet
  (move
    (dir north)
    (on (sidewalk
          (street elm)
          (side east)
          (block 100)))
    (to (sidewalk
          (street birch)
          (side north)
          (block 100)
          (along 1))))
  ((facing north)
   (moving #t)
   (mov-speed fast)
   (loc (intersect
          (street1 elm)
          (side1 east)
          (block1 100)
          (along1 99)
          (street2 birch)
          (side2 south)
          (block2 100)
          (along2 5))))
  ((cross-street north)))
```

Figure 4.3: Reactive planlet index when situation involves moving across a street

rough coordinate system, X and Y values for the starting location and for the ending location. This avoids the pitfalls inherent in the location description format, which permits the same location to be described in multiple ways.

In the next sample run, the Indexer does retrieve an indexing rule from memory. The indexing rules are applied to the basic index, and alter it to include the new features and information:

```
Rob(CBR)>> New destination is:
The west side of oak, the 200 block, 50 feet along
Rob(Ind)>> Creating plan index
Rob(Ind)>> Creating index rule index          Recursive application
Rob(Ind)>> Index is:                          of Indexer to find
(indexer plan 119 80 117 190)                 specializations of the
Rob(Ret)>> Starting to retrieve               basic index
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case indexerg1929 matched with difference value of 5
Rob(Ret)>> Cases under consideration are:
  (indexerg1929)                              One indexing rule applies

Rob(Ind)>> Index is:
(plan (sidewalk (street oak)                  Starting location
                (side west)
                (block 100)
                (along 60)
                (across 1))
      (sidewalk (street oak)                  Goal location
                (side west)
                (block 200)
                (along 50)
                (across 3))
      (*spec-index* both-stay-on-x 1928))     Special feature from rule
```

Because this indexing scheme requires using the Indexer and Retrieval components recursively, the potential exists for endless iteration of indexing and retrieval. In specializing an index the Indexer needs to create an index; in creating that index it may want to specialize it, which leads to another index creation, and so on. To eliminate this potential problem for the time being, we chose to restrict indexing rules in memory to those created by introspective reasoning for application to `plan` indices. No other kind of index is specialized, therefore the Indexer will call itself at most once. This is a somewhat ad hoc solution, but a more principled solution is not obvious.

If the Indexer could specialize any kind of index and so end up with unlimited recursive calls to itself, it must be able to determine when to *stop* trying to specialize an index. One alternative solution, perhaps equally ad hoc, would be to limit the number of recursions of the Indexer. A better solution might be based on the *reason* for specializing the index. For indices to retrieve plans, we want specialization to make sure that relevant features are included: so that the best, most adaptable plan is selected. For an `indexer` index (the only kind we need consider for recursive Indexer calls), specialization might serve to expand the indexing rules which are applicable to a given situation. In this case, the Indexer could choose to specialize non-`indexer` indices automatically, but would only specialize an `indexer` index if retrieval with the basic `indexer` index failed to discover any matching rules. If no matches are found after several recursive iterations, the process could be halted and the assumption made that no rules applied to the original specialization request.

The final index is the result of the basic index creation method, and any changes made by indexing rules. The result is passed along to the Retriever for selection of cases of the appropriate type and number from memory.

Plan cases	Differences between slots in starting locations, and in ending locations, normalized for `sidewalk/intersect` differences, plus differences between special indices, regardless of order
Indexing rules	Unification with variable binding, where numbers within 15 of each other are taken to match
Adaptation strategies	Unification with variable binding
Execution planlets	Unification with variable binding on step and location parts of index, unification of elements of "news" list, regardless of order in list

Table 3: Methods for similarity assessment

4.4 Retriever: Selecting cases from memory

ROBBIE's memory contains four different kinds of memory structures; the Retriever must satisfy the needs of each different kind of retrieval. It does so using a single general process with individualized measures of similarity and different retrieval "modes" which describe restrictions on the number of cases to be returned.

The Retriever needs to be able to evaluate the relative similarity of cases to the current situation, and to rank those cases against one another. In comparing two indices, the Retriever generates a *difference value* which quantifies how similar or different the indices are. The value is zero if the indices match exactly and increases as the indices become less similar, up to a pre-defined maximum value when two cases are incompatible, as when indices refer to different kinds of memory structures (e.g., `plan` and `adapt`).

Different memory structures require different measures of similarity because their indices describe different kinds of features. Plan indices contain location descriptions whose slots may be compared to other location descriptions. Indices for other memory objects often include variables which may be matched to the current index with unification. Table 3 describes the similarity measure for each kind of memory object.

Indices for retrieving plan cases are compared by taking a weighted sum of each difference in the three parts of the plan index: the starting location, ending location, and special features. A difference for the location portions of the index is defined as any place where the fillers of a particular slot are incompatible (after accounting for different possible location descriptions of the same location). A difference for special features occurs when a feature exists in one index and does not exist in the other. Other memory objects are chosen by unification of the current situation index with the case's index, further discriminated by the number of variable bindings that were made. Since variables may appear in either the case in memory or in the current situation index, fewer variable bindings often means a closer match.

Besides using different similarity measures for different kinds of cases, the Retriever may return differing numbers of cases as matches to the current situation, depending on the needs of the component that calls it. The component of the system invoking the Retriever provides it with the appropriate index for the current situation, and also describes the needed retrieval mode. The Retriever may return only one case, or it may return many. It may permit no cases to match, or it may consider no matching cases a failure. If the object is a plan or an adaptation strategy, exactly one case must be selected and returned. The other memory structures (indexing rules and reactive planlets) require all cases matching within a pre-determined level of similarity to be returned. Because no indexing rules may apply to a given situation, it is acceptable for an `indexer` retrieval to discover no matching cases, but requests for other kinds of cases must result in the retrieval of at least one case matching the given index.

ROBBIE is unique in incorporating adjustable similarity and retrieval modes into one retrieval mechanism: the kind of retrieval and similarity assessment is controlled by the component calling for retrieval. Each component can specify, along with the index describing its knowledge needs, whether it needs one case or many, and whether or not no matching cases is permissible.

To ensure that it finds at least one match (when that is required) while limiting the number of cases considered similar at any one time, the Retriever may vary how high a difference value will be considered "close enough" to include a case in its pool of possible matches. If no cases match with the most restrictive similarity measure, the Retriever tries again with less selective measures, until it cannot be less selective, or it finds at least one matching case. If there are many similar cases in memory, only the most similar will match under the restrictive measure and be considered. If few cases match closely, lessening the restrictiveness will allow consideration of cases which would otherwise be discarded.

In retrieving cases, the Retriever considers each case in turn, calculating a difference value for the case's index and the current goal index.[5] The Retriever builds a pool of those cases whose indices matched within the current selectivity level. If a single case is to be retrieved, the pool of possibilities is examined more closely to select the case with the lowest difference value. If more than one case has an equally low difference value, a single one is chosen at random. If multiple cases are desired, the entire pool is returned. This is similar to the *validated retrieval* approach used by Simoudis & Miller (1991) and Kolodner (1988) in which partially matching cases are collected on the basis of coarse-grained features and more fine-grained analysis makes the final determination. In ROBBIE's case, the final determination does little more analysis than the original, but explicitly compares partial matches to each other.

[5]A more sophisticated memory access method would permit more efficient retrieval, but access issues are not the heart of this research and are being addressed by many others elsewhere (Kolodner, 1993a).

```
(((across 3 unspec))      Differences in starting locations
 ((street elm birch)      Differences in goal locations
  (side east north)
  (block 200 100)
  (along 90 1))
 ())                      Differences in special features
```

Figure 4.4: Differences for case `old2` in first sample run

The sample output in the previous section showed the use of the Retriever to retrieve indexing rules. For the Indexer, if multiple rules had matched, even with varying difference values, they would all have been returned to the Indexer. Continuing the first sample run above from the point where the index is provided to the Retriever,[6] plan retrieval results in a random choice between two equally similar plan cases:

```
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3            Initial, most stringent level
Rob(Ret)>> Case old1 matched with difference value of 40
Rob(Ret)>> Case old2 matched with difference value of 25  old2 and old3
Rob(Ret)>> Case old3 matched with difference value of 25  match equally
Rob(Ret)>> Cases under consideration are:
  (old1 old2 old3)
Rob(Ret)>> Selecting case: old2            old2 is chosen randomly
```

The differences associated with case old2 are described in Figure 4.4. There are three sets of differences: differences in the starting locations (insignificant in this case), differences in the goal locations (quite significant), and differences in special features (none).

The cases retrieved may be used in very different ways, to be adapted if the

[6]The sample run in its entirety can be found in Appendix A.

case were a plan, to be applied as indexing rules or adaptation strategies, or to be evaluated and executed for reactive planning. The Retriever therefore returns a general retrieved-object structure (or a list of such objects) containing the case retrieved and the circumstances regarding its retrieval, which the various components must decode and use. Information in the retrieved-object structure besides the case itself includes the difference value calculated, and the differences themselves. The differences are often important for later application of the case, either in determining what to adapt in a plan, or using the variable bindings made in unification in the body of the case.

4.5 Adaptor: Altering cases

The Adaptor component of ROBBIE modifies retrieved cases to construct a solution for the current problem. The adaptation process maps directions and locations in the retrieved case onto the current situation and fills in the changed details. The Adaptor uses case-based retrieval to select adaptation strategies for specific portions of the new plan, or for supporting knowledge structures the Adaptor creates. It applies the selected rule to fill in the information needed at that specific point. The process is repeated until the new plan is completely adapted. The first sample run described in previous sections continues with adaptation:

```
Rob(Ada)>> Considering adaptation of plan old2
Rob(Ada)>> Plan needs adaptation to repair differences
Rob(Ada)>> Mapping for cases found, beginning adaptation loop
```

The first task in adaptation is to determine how the directions (north, south, east, and west) in the retrieved plan correspond to the directions of movement in the new situation. If memory is sufficiently large and evenly covers the world, it will be possible to retrieve plans whose directions match completely (moving east in the retrieved plan will correspond to moving east in the new plan). However, ROBBIE starts out with a very small initial case memory, and permits matching of two indices with any different directionality: a situation involving moving south and east may cause retrieval of a plan for moving north and east. Naturally, a case with the same directionality will be preferred in retrieval to one with different directionality and the same level of similarity in other features. The Adaptor determines how the cardinal directions map from old to new: for the previous situation it would map north onto south, south onto north, and east and west onto themselves. From this mapping, the Adaptor determines how to alter the directions in the old plan to fit the new one.

The Adaptor uses constraints on the current situation to determine what directions of movement could correspond to directions in the retrieved case. One sample constraint is that if the starting location is in the middle of a block, the first `move` must involve moving in the directions of that street: if the location is on an east/west street then the first move must be either east or west. If the retrieved case's first move is north, then the Adaptor can constrain the mapping to map north and south onto east and west (and vice versa). Other constraints will allow the mapping to be further specified until a single mapping is found.

Determining how directions of movement map from old to new situations is important as the basis for deriving a new plan from the old one, but it also serves to judge the adaptability of a case. A retrieved case which is inapplicable (either because of poor retrieval or differing levels of complexity) will be identified when the Adaptor cannot find a mapping for the directions of movement. If a case cannot be adapted, that fact is determined before much effort has been put into the adaptation process, and the planner can then choose to alter the goals it is trying to reach through re-retrieval.

If a mapping is found, as in the sample run above, the Adaptor continues by initializing the knowledge structures needed for deriving the new case. In order to adapt the old situation by mapping it onto the new one, it must make changes to the direction of movement of the case, but also to the locations in the case and their individual slot values. These changes are often dependent on one another; if the location in a `turn` step is altered, the location in the following `move` step should be affected. These slot-dependencies are best considered in isolation from other slots (i.e., sides of streets are considered separately from blocks of streets). Therefore, the Adaptor uses a set of supplemental knowledge structures which permit it to consider just those features of a plan step which are dependent on each other without continually accessing the complex plan step structures directly.

The main reason that supplemental knowledge structures are necessary emerges from the case-based approach to adaptation. In order to retrieve the appropriate strategy from the case memory, the Adaptor must be able to describe the current portion of the problem and the other features on which its value depends. This is difficult to do if those features are buried inside locations in other plan steps, but is easy when those features are immediately adjacent and available.

Supplemental knowledge structures are constructed during adaptation to describe the directions of movement for each plan step, and the streets, sides of streets, and actual locations for each *location* specified in the plan steps. Each structure is initialized with the values that are known when adaptation begins (at least the starting and ending values which come from the plan index itself). The rest of the values are

```
((starting-at (maple))              ((starting-at (maple))
 (turn empty)                        (turn (maple))
 (move-on empty)                     (move-on (maple))
 (move-to empty)           =>        (move-to (elm))
 (turn empty)                        (turn (elm))
 (move-on empty)                     (move-on (elm))
 (move-to empty)                     (move-to (elm))
 (ending-at (elm)))                  (ending-at (elm)))
```

Figure 4.5: Initial and final forms of `new-streets` supplemental Adaptor knowledge

marked as `empty`, indicating to the Adaptor that they have not yet been filled in.

The distinction between locations and plan steps is important, since `move` steps involve two locations with two different meanings: one location *on* which the move takes place, and another *to* which the move is headed. Separating the features of these two locations into equally accessible members of a list is important for determining correct adaptation strategies to apply to each slot. Figure 4.5 shows the initial and final forms of the `new-streets` knowledge structure for the first sample run we described. The `new-streets` structure describes the streets for each location in the plan being adapted.

Corresponding to each knowledge structure like `new-streets` which describes the values of a particular class of knowledge for the new plan being created, a separate knowledge structure describes in similar terms the values for the original plan. To complete the adaptation process, a skeletal template for the new plan is created which has the same form as the retrieved plan. Each of its parts requiring adaptation is also filled with `empty` markers. These empty markers are then filled in from the supplemental knowledge structures as the information becomes available.

Once the knowledge structures for adaptation are initialized, the Adaptor repeatedly selects and applies adaptation strategies until the new plan is completely adapted (there are no `empty` markers left). At each cycle, the Adaptor selects a spot in one of the knowledge structures which contains an `empty` marker. Initially, the spot to be adapted is chosen randomly from either the skeletal plan or the structure for directions of movement, `new-dirs`. These two adaptation structures are singled out because many pieces of the skeletal plan may be filled in directly without reference to the supplemental knowledge structures, but most of the other supplemental structures depend on values in `new-dirs` being filled in before they can be adapted. Once the directions of movement are complete, all knowledge structures including the skeletal plan are options for adaptation. Which structure, and what portion of it, is chosen at random.

```
Rob(Ada)>> Mapping for cases found, beginning adaptation loop

Rob(Ada)>> Selecting next point of adaptation:
 (0 ((starting-at (dir any) (on empty))          Adapting skeletal plan,
     (turn (dir empty) (on empty))
     (move (dir empty) (on empty) (to empty))
     (turn (dir empty) (on empty))
     (move (dir empty) (on empty) (to empty))
     (ending-at (dir any) (on empty))))
 (1 (turn (dir empty) (on empty)))               The second step,
 (2 (on empty)))                                 The "on" location

Rob(Ind)>> Creating adaptor strategy index       Selecting a strategy:
Rob(Ind)>> Index is:
(adapt start copy (on empty) turn)
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case adapt5 matched with difference value of 0
Rob(Ret)>> Case adapt45 matched with difference value of 4
```

```
Rob(Ret)>> Cases under consideration are:
  (adapt5 adapt45)
Rob(Ret)>> Selecting case: adapt5                     If needed value is
Rob(Ada)>> Applying strategy:                         blank, do nothing,
(if (blank inter-val) (no-op) (value-of inter-val))  else fill in value:
Rob(Ada)>> NO-OPING                                   Rule does nothing
...
Rob(Ada)>> Selecting next point of adaptation:
...
Rob(Ind)>> Creating adaptor strategy index
Rob(Ind)>> Index is:
(adapt info new-dirs (index 7) major)           Adapting new-dirs
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case adapt45 matched with difference value of 4
Rob(Ret)>> Case adapt22 matched with difference value of 1
Rob(Ret)>> Cases under consideration are:
  (adapt45 adapt22)
Rob(Ret)>> Selecting case: adapt22
Rob(Ada)>> Applying strategy:                   Look up old direction
(map (look-up old-dirs ?ind))                   and map it to new value
...
```

It might seem easier to derive the fillers for each knowledge structure in turn: first the directions, then the streets, then the sides of streets, then locations, and so forth. Choosing at random does have the negative effect that empty slots are chosen which cannot be filled from the available information (as the example above shows). If it were easy to determine the correct order in which slots could be filled, then it would certainly be wasteful to use a random approach. However, the dependencies between different features are complex. We cannot start with one structure and fill it in completely before beginning on the next. The adaptation strategy `adapt39` in Figure 4.7 below is an example of a strategy in which the value of one slot (the street for a location) depends on the value in a knowledge structure which would seem to come later in the process (the list of complete new location descriptions).

```
(adapt37
  (adapt info new-streets turn ?ind filled blank filled)
  (strategy
     (value-of-all (prev-value new-streets ?ind))))
```

Figure 4.6: Adaptation strategy: fill in `turn` street with previous

Once an empty slot in the adaptation structures is selected, a description of it and its dependencies is passed to the Indexer, and the resulting index is used to retrieve an adaptation strategy from memory. The strategy is applied to alter the empty slot, if the right new value can be determined from existing knowledge. If not enough information has been filled in, the empty slot is left `empty` and will be reconsidered later.

Each adaptation strategy applies to a particular adaptive situation in which the system can infer new information from existing information. Figure 4.6 show a sample adaptation strategy. Its index says that it is an `adapt`ation strategy for supplemental `info`rmation, it affects the `new-streets` list with the street for a `turn` step at the `?ind` position in `new-streets`, where the preceding street is filled in, the current street is blank, and the next street is also filled in. The strategy is to look up the values of the preceding street in the sequence and fill in the current street with those values. A `turn` location could be preceded by a `move-to` location, another `turn` location, a `starting-at` location, or a `ending-at` location, but not by a `move-on`. Since the robot will not have moved between the ending of the preceding step and the current step, the `turn` must occur on the same street(s).

Figure 4.7 shows a few selected adaptation cases. Each case shares some features with other memory structures: a name, an index, and then whatever other information the case includes. The indices for adaptation strategies differ in the number of features each includes, but share a general structure. The first few features identify the case as an adaptation strategy, and then identify which of the adaptation knowledge structures is affected. The feature `start` refers to the beginning portions of the new plan, `inter` and `end` indicate the middle and ending portions of the plan, respectively. The feature `info` indicates a supplementary knowledge structure, and is followed by the name of the structure involved. Later parts of the index describe other relevant features, such as the kind of change to be made. The heart of the adaptation structure is the `strategy` which describes how to alter the affected knowledge structure. The strategy is described using a simple Scheme-like language which is interpreted by the Adaptor.

For adapting cases which may not be extremely close matches, ROBBIE must have access to more information about street locations than is explicitly available from cases in memory. The Adaptor must choose intermediate streets, when the plans call for them, which connect the endpoint locations of its new plan. ROBBIE has, in addition to its case memory, information about individual streets from which it can infer appropriate intermediate streets for plans (an appropriate intermediate would be one that intersects with both the starting and ending streets or, failing that, that intersects with at least one of them). Figure 4.8 shows an example of the typical information ROBBIE knows about a street. The street information includes which blocks it runs along, which direction it runs, and between which blocks it lies. From this information, ROBBIE can derive the set of streets which intersect with a given street, and at which blocks they intersect. This provides enough information to select appropriate intermediate streets.

Once all `empty` slots of the new plan have been filled in with values by adaptation strategies, the resulting plan is passed along to the Executor to be executed and evaluated. Before discussing execution, we will describe the re-retrieval process which

Fills in direction slot of a "starting" plan step:
```
(adapt1 (adapt start simple (dir empty) major)
        (strategy (map old-plan-value)))
```

Copies "on" location from location list to starting `turn` *step:*
```
(adapt5 (adapt start copy (on empty) turn)
        (strategy (if (blank inter-val)
                      (no-op)
                      (value-of inter-val))))
```

Fills in direction list element:
```
(adapt22 (adapt info new-dirs (index ?ind) major)
         (strategy (map (look-up old-dirs ?ind))))
```

Fills in new location in middle of plan by constructing from `street` *and* `side` *structures:*
```
(adapt23c (adapt info new-locs inter ?ind ?a blank ?b)
          (strategy
          (if (or (blank (look-up new-sides ?ind))
                  (blank (look-up new-streets ?ind)))
              (no-op)
              (construct-rep (look-up new-streets ?ind) ?ind))))
```

Narrows possible streets by intersecting current with next neighbor:
```
(adapt25 (adapt info new-streets move-to ?ind fixed filled filled)
         (strategy
           (find-inters
             (set-intersect (look-up new-streets ?ind)
               (next-value new-streets ?ind)))))
```

Fills in new street by calculating intersecting streets from previous location (street and side information required):
```
(adapt39 (adapt info new-streets move-on ?ind fixed blank ?any)
         (strategy
            (if (or (blank (prev-value new-locs ?ind))
                    (blank (look-up new-dirs ?ind)))
                (no-op)
                (find-inters
                  (rule1 (prev-value new-streets ?ind)
                         (value-of (prev-value new-locs ?ind))
                         (look-up new-dirs ?ind))))))
```

Figure 4.7: Selected adaptation strategies

```
(street (name maple)
        (extent 100 300)
        (runs e/w)
        (size small)
        (loc 000))
```

Figure 4.8: Street information for ROBBIE

allows ROBBIE to extend the complexity of plans with which it can work by altering its goals, subdividing complex problems into more simple pieces.

4.6 Reretriever: Altering the goal

It is possible that a retrieved plan will not be adaptable, either because it is too poor a match to the current situation or because the current situation requires a more complex plan than any case currently in memory. For instance, if all plans in memory contain only one step, a move in some direction to some point, then no plan exists which will solve a problem that requires turning a corner and moving in a new direction. ROBBIE recognizes these situations when the Adaptor cannot adapt the case which was retrieved as the best match in memory:

New sample run where re-retrieval is required.

```
Rob(CBR)>> New destination is:
The north side of maple, the 100 block, 50 feet along
Rob(Ind)>> Creating plan index
Rob(Ind)>> Creating index rule index
Rob(Ind)>> Index is:
(indexer plan 25 75 70 23)
...
Rob(Ret)>> Cases under consideration are:
  ()
Rob(Ind)>> Index is:
```

```
(plan (sidewalk (street elm)
                (side east)
                (block 100)
                (along 55)
                (across 5))
      (sidewalk (street maple)
                (side north)
                (block 100)
                (along 50)))
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Case old1 matched with difference value of 20
Rob(Ret)>> Case #:g1915 matched with difference value of 40
Rob(Ret)>> Case #:g1916 matched with difference value of 20
Rob(Ret)>> Cases under consideration are:
  (old1 #:g1915 #:g1916)
Rob(Ret)>> Selecting case: g1916                  Selects a learned case
Rob(Ada)>> Considering adaptation of plan g1916
Rob(Ada)>> Plan needs adaptation                  Case cannot be adapted
Rob(Ada)>> Plan couldn't be adapted, trying re-retrieval
```

The Reretriever alters the goal of ROBBIE to make the current problem more tractable by breaking it into two parts and recursively applying case-based planning to solve each part separately. An intermediate location is chosen between the current starting and goal locations:

```
Rob(Reret)>> General dir = (east south)          Route lies south and east
Rob(Reret)>> Options are = (maple birch)         Neighboring streets
Rob(Reret)>> Runs = n/s                          are north/south
Rob(Reret)>> Choosing intermediate location
(intersect (street1 elm)
           (side1 east)                          Intermediate location
           (block1 100)                          is on nearest corner of
           (along1 3)                            starting street and
           (street2 maple)                       neighbor to the east
           (side2 north)
           (block2 100)
```

```
          (along2 unspec))
```

A route plan is created from the starting location to the intermediate one:

```
Rob(Reret)>> First new index:
((sidewalk (street elm)                                Old starting location
           (side east)
           (block 100)
           (along 55)
           (across 5))
 (intersect (street1 elm)                              Intermediate location
            (side1 east)
            (block1 100)
            (along1 3)
            (street2 maple)
            (side2 north)
            (block2 100)
            (along2 unspec)))
Rob(Ind)>> Creating plan index
...
Rob(Ret)>> Selecting case: g1916
Rob(Ada)>> Considering adaptation of plan g1916
Rob(Ada)>> Plan needs adaptation
...
```

and another route plan is created from the intermediate to the goal.

```
Rob(Reret)>> Second new index:
((intersect (street1 elm)                              Intermediate location
            (side1 east)
            (block1 100)
            (along1 3)
            (street2 maple)
            (side2 north)
```

```
            (block2 100)
            (along2 unspec))
 (sidewalk (street maple)                        Old goal location
           (side north)
           (block 100)
           (along 50)))
Rob(Ind)>> Creating plan index
...
```

The plans are then concatenated to form a single route:

```
Rob(Reret)>> New steps are:
((starting-at (dir any)                 Steps from first plan start here
   (on (sidewalk (street elm)
                 (side east)
                 (block 100)
                 (along 55)
                 (across 5))))
 (turn (dir south)
       (on (sidewalk (street elm)
                     (side east)
                     (block 100)
                     (along 55)
                     (across 5))))
 (move (dir south)
       (on (sidewalk (street elm) (side east)))
       (to (intersect (street1 elm)
                      (side1 east)
                      (block1 100)
                      (along1 3)
                      (street2 maple)
                      (side2 north)
                      (block2 100)
                      (along2 unspec))))
 (turn (dir east)                       Steps from second plan start here
       (on (intersect (street1 elm)
                      (side1 east)
                      (block1 100)
```

```
                      (along1 3)
                      (street2 maple)
                      (side2 north)
                      (block2 100)
                      (along2 unspec))))
(move (dir east)
      (on (sidewalk (street maple) (side north)))
      (to (sidewalk (street maple)
                     (side north)
                     (block 100)
                     (along 50))))
(ending-at (dir any)
  (on (sidewalk (street maple)
                 (side north)
                 (block 100)
                 (along 50)))))
```

In theory, the re-retrieval process could be applied recursively and indefinitely. Once again the spector of infinite regress appears: ROBBIE could continuously choose intermediate locations which lead to unadaptable retrievals. If intermediate locations always simplify the route, however, and if retrieval is performing sufficiently well, the Reretriever should eventually retrieve a simple and adaptable case. We have restricted re-retrieval to non-recursive functioning, because of limitations to the heuristic for selecting intermediate locations: if one of the new subgoals retrieves an unadaptable case, ROBBIE simply gives up on the current goal.

The selection of the intermediate location is the key to a successful reformulation of the problem. We chose a straightforward and simple method for selecting the intermediate location in order to avoid extensive reasoning overhead. The intermediate location is chosen as one of the street corners closest to either the starting or goal locations. Which original location to work from is chosen at random. ROBBIE can determine the general direction in which it needs to move to solve a given problem

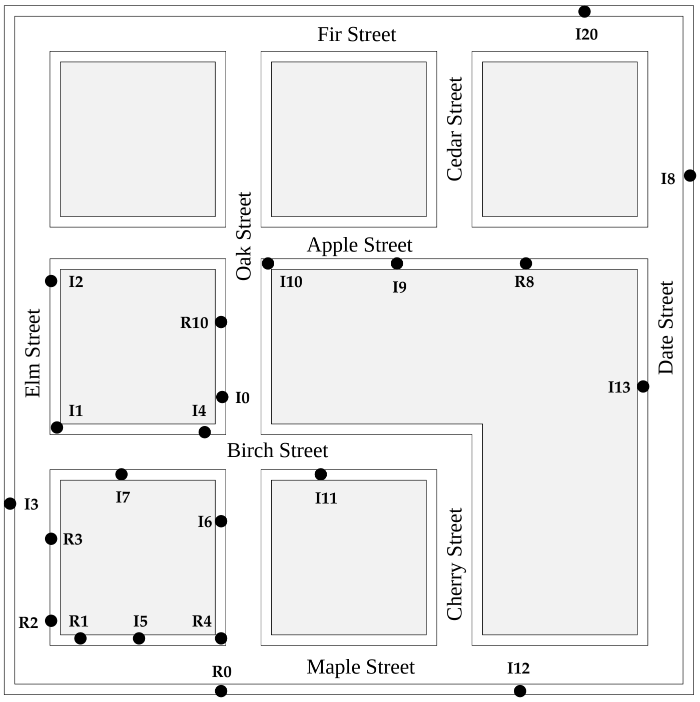

Figure 4.9: Map annotated with locations

(i.e., north and east) from its knowledge of where streets lie in the world. It selects an intersection one block from the selected location of the original problem in one of the directions the problem requires moving in. For example, in Figure 4.9,if the starting location for a problem were at **R4**, and the goal were at **I8** so that the direction of movement would be north and east, the intermediate location starting from **R4** would be either "Birch and Oak" or "Maple and Cherry." This method is not foolproof, consider the starting location **I11** and the goal location **R8**. The likely intermediate location starting from **I11** would be the corner of "Birch and Cherry," which would probably make the situation worse. Still, an occasional failure due to re-retrieval is better than always failing when an unadaptable case is retrieved, which is the situation without re-retrieval.

ROBBIE occasionally retrieves an unadaptable case even when other cases exist in memory which would be adaptable (due to its limited initial indexing scheme). Re-retrieval may be applied in this case, as when the problem is due to plan complexity, but the failure to retrieve the right plan should be considered a reasoning failure. In later chapters we will describe how introspective learning can overcome such reasoning failures by avoiding poor retrievals.

Once an intermediate location is selected, the Retriever and Adaptor are called recursively to create routes for the two new goals. Since the resulting routes share a middle location, their steps can be concatenated to create a new plan to solve the whole original problem.

4.7 Executor: Reactive execution

A plan which has been retrieved and adapted is presumed to be a correct solution to the given problem. That presumption, however, must face the acid test of actual execution of the plan. While we presume the plan to be correct, we do not assume it to be correct, and so ROBBIE includes a reactive execution component which tests out the new plan in its simulated world and learns from any problems that might arise.

ROBBIE's Executor considers each plan step created by case-based planning as a goal to be achieved in sequence. It does so using a variation of reactive planning (Firby, 1989; Freed & Collins, 1994a; Agre & Chapman, 1987; Brooks, 1987).

Plan steps are stated as commands ("Turn east on Maple and Oak"), but within each step is an implicit goal to be achieved. The steps `starting-at` and `ending-at` encode the implicit goal that the current location match[7] the location in the step

[7]We will discuss below what it means to the planner for locations to match.

without further action by the reactive planner. A `turn` step encodes the goal that the system take actions which result in the simulated robot facing in the right direction (without changing the location of the robot). A `move` step describes the goal of taking actions which result in achieving a given location, preferably while staying on another given location.

The Executor's reactive planning approach is based on the idea of Reactive Action Packages (RAPs). A RAP is a collection of methods for solving the goal of the RAP, and each method is distinguished by the context in which it may be applied. In ROBBIE the individual reactive objects are *planlets* which correspond to individual methods for specific contexts, not collections of them. Planlets for the same goal share similar indices, and often are retrieved as a group: such a group corresponds to the set of methods within a single RAP. A planlet is stored in case memory, indexed by the general goal it achieves (the plan step), plus features from the sensory input to ROBBIE which are relevant to the method's context.

"Contexts," in this form of reactive planning, refer to the moment-to-moment state of the world: where the simulated robot is, what it is doing, and what other objects are nearby. When the context changes, as when the robot approaches a street corner, a new planlet is selected to respond to the new context.

Figure 4.10 shows a typical planlet from ROBBIE's case memory. The index of the plan contains a description of the plan step to which the planlet refers, a description of the robot's state and location (ROBBIE's derived description, not the world simulator's), and a list of the "news" items for the robot. "News" items are features of the current sensory input from the world which are somehow out of the ordinary or deserving of special attention. Expected sensory input such as the new status of the robot or normal nearby objects (streets and walls, unless they are very

```
(planlet2b
  (planlet (turn (dir ?dir) (on ?on))            Planlet index
           (?x ?y ?z ?q)
           ((cross-street ?dir)))
  (and (on ?on) (dir ?dir))                       Goal context
  (and (not (dir ?dir)) (cross-street ?dir))      Applicability context
  ((stop) (turn ?dir)))                           Commands of planlet
```

Figure 4.10: A sample planlet for executing a turn step

close), are not "news:" they are not special enough to be considered a context change. Extraordinary inputs which do constitute a possible context change include: colliding with a wall, being within a step of entering or exiting a street, and the state of objects which the simulated world was told to `look-at` (such as traffic lights).

Each planlet describes a method which may be applied in a particular context for achieving some goal. It must, therefore, include a description of the goal to be achieved, and the situation in which it may be applied. We consider both requirements as "contexts" which may be described using the same vocabulary (a simple predicate form). The first context in the planlet structure describes when the planlet's goal has been achieved: for the example in Figure 4.10 the goal states that the robot will be located on the `on` location of the plan step, and will be facing the direction given in the `turn` step. The second context determines when the planlet is applicable. The planlet `planlet2b` applies when the robot is not already facing the right direction, and when a street is about to be entered in the goal direction.

The last portion of a planlet is the list of commands which describe the appropriate response to the current situation and goal. In `planlet2b` the correct response is to stop moving, and then turn in the desired direction. By stopping *before* turning, the robot will be sure not to accidentally step into the street which lies in that direction.

The Executor maintains an execution log describing each planlet that is applied. From that log ROBBIE reconstructs the actual plan that was needed to reach the goal: hence reactive execution also acts as a last stage of adaptation.

When the Executor is just starting a plan, or when it has just finished one plan step and is starting the next, it retrieves applicable planlets from memory and selects the right one by its context:

```
Time = 5:00:01
...                               Plan created here (first sample run above)

Rob(Exe)>> No current step, getting from plan
Rob(Ind)>> Creating planlet index
Rob(Ind)>> Index is:
(planlet
  (starting-at (dir any)
    (on (sidewalk (street maple)
                  (side north)
                  (block 100)
                  (along 20)
                  (across 3))))
  ((facing east)
   (moving #f)
   (mov-speed none)
   (loc (sidewalk (street maple)
                  (side north)
                  (block 100)
                  (along 20)
                  (across 3))))
  ())
Rob(Ret)>> Starting to retrieve              Retrieving planlets
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case planlet0a matched with difference value of 10
Rob(Ret)>> Case planlet0c matched with difference value of 10
Rob(Ret)>> Cases under consideration are:
  (planlet0a planlet0c)                      All matches returned
Rob(Exe)>> Choosing between planlets:
```

```
(planlet0c planlet0a)
Rob(Exe)>> Selecting planlet: planlet0a        Selected by applicability
Rob(Exe)>> Executing starting-at

Rob(Exe)>> Come to a stop                      Planlet says to stop moving

Robot is currently stationary...
Robot is facing east at the position 20 feet along maple street
```

Often the index for retrieving planlets selects exactly one method which applies to the current situation. However, some contexts require more information than the indices contain. Hence the index retrieves all the planlets which might match the current situation given information in the index, but the context is required to make the final discrimination (determining whether the robot is "on" or "to" a given location is one task which cannot be done by the current indexing scheme). Once a group of planlets has been retrieved, therefore, their contexts are checked to determine which, if any, apply to the current step.

As in Firby's model, before selecting a planlet, the goal context is evaluated to see if the goal of the plan step has already been achieved. If the goal context is true, then the methods retrieved do not need to be applied, and the Executor can move on to the next plan step. In this way redundant or otherwise unnecessary plan steps will automatically be pruned from the final plan: the planlets retrieved when initially considering a new plan step are not stored in the execution log until they are *applied*.

If the goal is not already achieved, the Executor evaluates the context in which each retrieved planlet applies, to determine which matches the current situation exactly. If more than one matches, the choice is made among them randomly.

After a planlet has executed for one time step, there are three possible outcomes. In the first case, the goal of the planlet is achieved, and the Executor moves on to the next high-level plan step:

```
Time = 5:00:02

Rob(Exe)>> Step successful!!                    Previous planlet's step done
Rob(Ind)>> Creating planlet index               Executor goes on to next step
...
Rob(Exe)>> Choosing between planlets:
(planlet2 planlet1)
Rob(Exe)>> Selecting planlet: planlet2
Rob(Exe)>> Executing a turn to the west         Executes next step

Rob(Exe)>> Come to a stop
Rob(Exe)>> Turning west

Robot is currently stationary...
Robot is facing west at the position 20 feet along maple street
```

If the goal of the current plan step has not been achieved, the Executor must choose whether to continue applying the method from the previous step, or to retrieve a new method. Whether the Executor chooses to continue its current planlet or select a new one depends on changes in the information coming from the simulated world. If the sensory input for a new time step contains anything the Executor classifies as "news," the Executor will retrieve a new method to respond to it. If the context remains the same as the previous time step, the Executor will continue to apply the currently selected planlet:

```
Time = 5:00:03
...          Turn from previous step was successful, going on to a move

Rob(Exe)>> Executing a move on maple
```

```
Rob(Exe)>> Move along

Robot is currently in motion...
Robot is moving west to the position 17 feet along maple street

------------------------------------------
Time = 5:00:04

Rob(Exe)>> Continue current step    No context change, continue move planlet

Rob(Exe)>> Move along
...
```

The most common context changes the Executor encounters occur when the robot is very close to its goal location (and should slow down), or when it comes to a street it must cross:

```
...    Much time passes in this sample run
------------------------------------------
Time = 5:00:41

Rob(CBR)>> Close to entering street north            "News" from sensors

Rob(Exe)>> Context has changed, reconsidering planlet  Needs new planlet
Rob(Ind)>> Creating planlet index
Rob(Ind)>> Index is:
(planlet
  (move (dir north)
        (on (sidewalk (street elm) (side east)))
        (to (sidewalk (street elm) (side east)
                      (block 200) (along 90))))
  ((facing north)
   (moving #t)
   (mov-speed fast)
   (loc (intersect (street1 elm)
                   (side1 east)
```

```
                    (block1 100)
                    (along1 99)
                    (street2 birch)
                    (side2 south)
                    (block2 100)
                    (along2 5))))
  ((cross-street north)))            Index includes news: street to north
...
Rob(Exe)>> Choosing between planlets:
(planlet3h planlet3g planlet3d planlet3c
 planlet3f planlet3b planlet3a planlet3)
Rob(Exe)>> Selecting planlet: planlet3g    Planlet handles street-crossing
Rob(Exe)>> Executing a move on elm

Rob(Exe)>> Storing current step            To cross street, store the move
Rob(Exe)>> Making sub-goal: cross          step, add a new step that waits
Rob(Exe)>> Come to a stop                  for a green light, stop moving,
Rob(Exe)>> Looking at stop-light           and start watching the stop-light

Robot requests a look at upcoming stoplight

Robot is currently stationary...
Robot is facing north at 99 feet along and 5 feet across
the sidewalk at elm and birch
```

When all plan steps have been achieved, the simulated robot is at the goal location and the Executor's task is complete. The execution log is passed to the Storer so that a new case may be learned.

4.7.1 Matching actual to goal locations

In evaluating the contexts of a planlet, the Executor must determine whether or not locations match one another. Matching in this case does not imply an exact structural or content match, but something much more loosely defined. Figure 4.11 contains a set of location descriptions that all "match" from the Executor's perspective. The

```
(sidewalk (street birch)      (intersect (street1 birch)
          (side north))                  (side1 north)
                                         (block1 100)
                                         (along1 99)
(sidewalk (street oak)                   (street2 oak)
          (side west)                    (side2 west)
          (block 200)                    (block2 200)
          (along 3))                     (along 5))
```

Figure 4.11: Location descriptions that match, to the Executor

first general heuristic for matching is that locations need only match the features which are specified in the *goal*: a `sidewalk` description that only specifies a side of a street matches an `intersect` or `between-blocks` description which shares that street and side. A more fully specified location enforces a closer match. The second heuristic is that locations must match up to the `along` slot of the *robot's* location. If the goal location is in the middle of a block, the Executor insists that `along`, but not `across`, slots match exactly. If the goal is at a corner, then the actual location must match exactly *in the direction the robot is moving*, regardless of the value in the other direction. With this method of matching locations, the robot will get as close as possible to the specific location it is given, without having to include extra moves which simply shift its position sideways to achieve an exact match.

4.7.2 Planlet actions

Each planlet contains a set of commands to ROBBIE and to the world simulator. Table 4 shows the set of commands planlets can currently make. Other than the normal "motor" and attentional commands to the world simulator, planlets can indicate when the robot encounters problems requiring re-planning, can record a step that was

`stop`	To simulator to stop robot
`move`	To simulator to move robot
`slow`	To simulator to move slowly
`speed-up`	To simulator to move quickly
`turn`	To simulator to turn robot
`look-at`	To simulator to look at object
`look-away`	To simulator to stop looking
`replan`	To Executor to solve planning problem
`record-turning`	To Executor to record a turn step
`store-current-step`	To Executor to put step back in plan
`make-new-step`	To Executor to add new step to plan

Table 4: Commands in planlets to Executor and world simulator

not a part of the original plan into the execution log, can push the current high-level plan step back into the queue of steps to insert a new step, and can create a new step to be inserted into the queue at the current spot.

The Executor is sufficient to execute correct plans well, given the sort of obstacles which occur in the world simulator. Figure 4.12 shows the three planlets which permit the Executor to cross streets with the green light. The first notices that the robot is about to enter the street, stops, and inserts a new goal to cross the street when the light is green. The second remains stationary until the light is green in the right direction. The third restores normal activity and starts the robot moving across the street. New planlets would be required to successfully maneuver new obstacles such as pedestrians and cars. In addition, the Executor's "re-planning" capabilities are currently limited to informing the introspective reasoner of a problem and giving up. If execution succeeds, the execution log of planlets applied is passed to the Storer component for reconstruction and addition to the case base.

```
(planlet3g
   (planlet (move (dir ?dir) (on ?on) (to ?to))
      ((facing ?dir) ?x ?y (loc ?loc)) ((cross-street ?dir)))
   (and (on ?on) (to ?to))
   (and (not (close ?to))
        (and (cross-street ?dir) (not (stop-light ?s1 ?s2 green ?dir))))
   ((store-current-step)
    (make-new-step cross ?loc ?dir)
    (stop)
    (look-at stop-light)))
```

Applies when a move in a given direction is barred by a street in that direction, and the stop-light is not green in the right direction. Puts the current plan step back into the queue of steps, creates a new step to cross the street, and then sets up state of robot correctly

```
(planlet5a
   (planlet (cross ?s1 ?s2 ?dir) (?x (moving #f) ?z ?q) ((cross-street ?dir2)))
   (stop-light ?s1 ?s2 green ?dir)
   (cross-street ?dir2)
   ((stop)))
```

Applies when the current plan step is a `cross` *(created by previous planlet). Makes sure the robot remains stationary as long as the stop-light is not green the right way. The goal of the step is achieved when the light is green.*

```
(planlet3h
   (planlet (move (dir ?dir) (on ?on) (to ?to))
      ((facing ?dir) ?x ?y (loc ?loc)) ((cross-street ?dir)))
   (and (on ?on) (to ?to))
   (and (not (close ?to))
        (and (cross-street ?dir) (stop-light ?s1 ?s2 green ?dir)))
   ((look-away stop-light)
    (move)))
```

Applies when a move in a given direction is barred by a street, and the stop-light is green that direction. Stops looking at the stop-light and simply moves into the street

Figure 4.12: Planlets for crossing streets

4.8 Storer: Adding cases to memory

The Storer component must piece together the final solution from the execution log, creating a normal high-level plan case, and store the resulting case in memory. Storing new solutions is the basis for domain learning in a CBR system. This learning improves ROBBIE's performance by widening the pool of potential matches for a new situation, and by diminishing the work required to adapt a dissimilar original plan when a similar learned one may be applied instead.

The Storer is also important because only after execution of a plan and the Storer's subsequent reconstruction of a final solution from the execution logs, is a definitive final solution for the current route-planning problem available. The existence of the final solution enables additional expectations of the introspective reasoner to be examined to monitor the planner for failures. ROBBIE has expectations about what the final solution should be in terms of what was retrieved, and the other cases in memory. When these expectations fail, ROBBIE learns to refine its retrieval criteria.

Execution logs contain a list of every planlet applied in executing the case-based route plan. Reconstructing a high-level plan from these small steps involves converting `starting-at`, `ending-at`, and `turn` planlets into their high-level equivalents, removing redundant turn steps (for example, when two turns are adjacent, the first has no ultimate effect on the plan solution), and compressing the multiple planlets usually required to achieve a `move` step back into a single high-level `move` step.

The resulting plan steps are inserted into a plan case structure, along with the appropriate index, and then stored in memory. The Storer should avoid storing duplicate cases: in adding a new case the Storer compares it to each old case to make sure there is no duplicate. If no match appears, the case is added to memory, otherwise no case is stored and no learning takes place. The planning process then repeats from the beginning:

```
Time = 5:01:19

Rob(Exe)>> Step successful!!                 Move step successful
Rob(Ind)>> Creating planlet index
...
Rob(Exe)>> Executing ending-at

Rob(Exe)>> Come to a stop

Robot is currently stationary...
Robot is facing north at the position 90 feet along elm street

-----------------------------------------
Time = 5:01:20

Rob(Exe)>> Step successful!!                 Last plan step
Rob(Exe)>> Plan complete
...

-----------------------------------------
Time = 5:01:21

Rob(Sto)>> Reconstructing finished plan
Rob(Sto)>> Irrelevancy: eliminate street crossing
Rob(Sto)>> Steps need combining
Rob(Sto)>> The plan to be stored:
   Omitted for space, see Appendix A

 The next output is triggered by the introspective reasoner.
It uses as an index the plan steps of the final solution to retrieve
the case in memory with the most similar plan steps.  If this
is not the case originally retrieved, a reasoning failures is
indicated: the retrieved case is not the closest case in memory.
Rob(Ind)>> Creating plan solution index
Index is:
...
```

```
Rob(Ret)>> Case old1 matched with difference value of 44
Rob(Ret)>> Case old2 matched with difference value of 14
Rob(Ret)>> Case old3 matched with difference value of 36
Rob(Ret)>> Cases under consideration are:
  (old1 old2 old3)
```
No introspective reasoning comment here means the best case was originally retrieved. No reasoning failure occurred.
```
Rob(Sto)>> Storing plan in memory under name: g5448
```

Other than plans, which are routinely added to memory, the other memory structures (e.g., indexing rules, adaptation strategies, or planlets) are only added through introspective reasoning. The introspective reasoner inserts new indexing rules into memory directly, bypassing the Storer.

4.9 Summing up

ROBBIE's planning module combines case-base planning — to create high-level descriptions of routes for getting from one place to another — with reactive planning to execute the high-level plan in ROBBIE's world to determine if the plan is correct, and to make any last adjustments to the plan to apply it to the world. This combination of case-based and reactive planning is one exceptional aspect of the planner. Another is the use of re-retrieval, which adapts the goals of ROBBIE to make them more manageable, when retrieved cases cannot be adapted to fit the current problem. A central design choice with broad ramifications is the use of ROBBIE's own case-based retrieval process to implement other components of the case-based reasoning process. Re-use of case-based retrieval actually simplifies the planner by avoiding multiple mechanisms for tasks which have different purposes but similar methods. The idea of re-using the same mechanism is a powerful one which may be extended

- ROBBIE begins with a small set of route cases, modeling the learning of a person who starts out relatively unfamiliar with the environment in which she finds herself.
- ROBBIE combines deliberative and reactive planning, getting the benefits of deliberation through case-based reasoning and the benefits of continual response to a changing environment through Firby-style reactive planning.
- Goals exist whose routes are more complex than any plan in ROBBIE's case memory. "Re-retrieval" permits ROBBIE to break a difficult problem into smaller pieces and solve each piece separately.
- ROBBIE uses its indexing and retrieval processes recursively to implement components of the case-based planner. The Indexer itself, the Adaptor, and the Executor all utilize structures stored in the same case memory from which the planner as a whole retrieves route plans.
- ROBBIE's general retrieval mechanism permits different kinds of indices, different measures of similarity, and different modes of retrieval.

Figure 4.13: Key features of ROBBIE's planner

to other memory objects and components in the future. It raises interesting issues about extending the re-use of case-based processes to adaptation of multiple types of memory structures, and learning of other memory structures as well.

Many deliberative planners expend great effort constructing a plan to solve a problem, and assume that that plan will be correct, so that execution will succeed by blindly following the described plan steps. While this is short-sighted, it is equally short-sighted to claim that there is no need to look ahead or consider the big picture in planning; that the best approach is simply to start taking actions in the world and worry only about the current situation. We have shown that deliberative and reactive planning can be successfully combined to gain the advantages that each has to offer. Our case-based deliberative planner does consider the big picture and develops an abstract set of instructions for achieving its goal. Execution of the plan is not blind, but rather acutely aware of the changing circumstances in which the simulated robot finds itself. Therefore, the final solution is the conjunction of deliberative preparation and externally-driven responses.

Chapter 5

Model-based Introspective Reasoning

> ROBBIE's model-based framework for introspective reasoning supports both detection and explanation of failures, and contains generic building blocks for constructing models of other systems. We describe the issues to be addressed in designing such a framework and our design decisions.

The overarching goal of examining model-based introspective reasoning is to develop an introspective framework for detecting, analyzing, and repairing reasoning flaws. Our approach addresses the problems of detection and analysis of failures in detail and, in principle, the problem of alteration of the underlying reasoning process as a general task. However, our research is particularly concerned with a single class of repairs which is particularly important for case-based reasoning systems: refining the indexing criteria used to retrieve cases. In this chapter we focus on the problem of *model-based* introspective reasoning in general. The next chapter will address how ROBBIE implements model-based introspective reasoning and applies it to introspective index refinement.

The task of our introspective reasoning system is to diagnose and repair reasoning failures of its companion performance system. To perform this task the introspective reasoner must be able to detect possible reasoning failures, to explain the detected problem in terms of the system's reasoning processes, and to alter the reasoning processes to repair the flaw and prevent future re-occurrences. In designing our introspective reasoning framework, and implementing it in ROBBIE, we were guided by a set of goals which we believe an introspective reasoning system ought to achieve. Those goals include triggering introspective learning in response to failures, giving equal weight to the task of detecting failures and the task of explaining their causes, continuously monitoring for failures during the reasoning process itself, and incorporating enough generality to ensure the framework could be applied to many different underlying reasoning systems. We elaborate on each aspect below.

Failure-driven learning: A central goal for this framework is for the introspective component to trigger learning by detecting reasoning failures. In general, failures point out precisely those places where the system most needs learning: places where its current knowledge or reasoning is inadequate. Failures are therefore viewed as opportunities for learning which will improve the performance of the system (Leake, 1992; Krulwich et al., 1992; Hammond, 1989; Ram, 1989; Schank, 1986; Riesbeck, 1981). Focusing on failures simplifies the introspective reasoner's task, and concentrates effort on those places where opportunities to learn are most strongly indicated. If the underlying reasoning appears perfect, there is no reason to expend effort second-guessing it.

Combining detection and explanation of failures: A second goal for this introspective reasoning framework is to incorporate in one set of knowledge and mechanisms both *detection* of failures and *explanation* of failures. Other approaches have focused on one task at the expense of the other; they often assume failures will be obvious and detectable by the performance system and hence monitoring for failures is unnecessary. We argue that a good introspective reasoner must be capable of performing both tasks well. Many potential failures involve *poor but successful* processing which in the long run will negatively affect the performance of the system. The introspective diagnosis system should detect such "hidden" failures and correct them, a task difficult for the performance system itself to address.

Continuous monitoring: A refinement of the previous goal is for the detection of reasoning failures to occur continuously during the reasoning process itself, as opposed to examining a completed reasoning trace after the fact, or off-line. Tying the introspective and underlying reasoning processes together keeps us honest about the costs of introspective reasoning, and also provides the opportunity for introspective learning to overcome otherwise insurmountable obstacles the system might encounter. For example, if the Executor failed to find any planlet matching a given situation because of poor indexing criteria, introspective learning might correct the faulty indexing and permit the planner to find the appropriate planlet and continue execution. Without introspective learning, the planner would be unable to recover.

Generality: We desire a framework which is general enough to be used with many different underlying reasoning systems, but which allows for enough specificity in the knowledge of the introspective reasoner to perform its task. The introspective knowledge about a particular reasoning process will differ from one system to another.

In order to apply a standard framework to build a model for that differing information, we must conceptually separate the *content* of the introspective model from the *terms* in which it is described. We must have a generic vocabulary in which to describe individual statements about the reasoning process; we must have building blocks which describe in general terms how reasoning processes are organized in order to structure the introspective knowledge of a given system's model. The knowledge of the underlying reasoning process should be represented declaratively, and should be manipulated by mechanisms independent of the content of the knowledge, except as that content is provided by the underlying system.

ROBBIE's introspective component uses model-based reasoning to detect and explain reasoning failures. As stated in Chapter 1, ROBBIE's introspective reasoning is performed using a declarative model of the underlying reasoning process which describes, in terms of explicit *assertions*, expectations about the ideal performance of the reasoning process. The assertions in the model provide expectations about reasoning performance which can be compared to the actual reasoning performance: an expectation failure occurs when an expectation is not true of the actual reasoning process. When an expectation failure is discovered, the system has an opportunity to learn. Failures are explained by searching the declarative model for other possible failures causally related to the detected one. The possibility of applying this approach for improving case-based reasoning systems was first proposed by Birnbaum et al. (1991), based on their earlier work involving self-debugging, chaining-based planners (Birnbaum et al., 1990).

Our approach to model-based introspective reasoning achieves all the goals described above. Detected failures drive the introspective learning task; without an expectation failure the introspective component takes no action. The model of the underlying reasoning process provides information for both monitoring for failures and explaining them. The structure of the model is designed to support continuous monitoring and evaluation of incomplete reasoning. The introspective reasoner uses mechanisms independent of the model's content to manipulate it and describes assertions using a vocabulary independent a particular system's features.

The introspective framework determines where to make the repair by explaining the detected failure, and can guide the repair process by suggesting potential classes of repair associated with particular assertions. For instance, if a poor case was retrieved and the retrieval process was known to be correct, then an obvious class of repair is to change the indexing criteria used to determine what to retrieve. The general problem of constructing a repair and implementing it remains open, but we describe our approach to one class of repair in the next chapter.

In this chapter we will describe our model-based introspective framework: what its assertions are, and how the model is structured. We will discuss how our design choices meet the goals described above, and discuss some general issues for introspective reasoning. In the following chapter we will discuss the application of these ideas to ROBBIE's index refinement problem.

5.1 Assertions of the model

In constructing the model of a system's reasoning processes, we must answer a set of questions about the contents of the model: which portions of reasoning process should be described by explicit assertions, in what detail the assertions should describe the reasoning process, and how assertions may be described in terms which are independent of any particular underlying system. In this section we will describe answers these questions in the context of ROBBIE and how those answers can be applied independent of any particular system.

5.1.1 What assertions to make

There is an apparent paradox in developing a model of the ideal reasoning process of a system. A completely detailed model of ideal reasoning processes would describe an implementation of that ideal. It would determine the correct conclusion for any situation. The difficulty of anticipating all situations and assuming correct behavior is, of course, what led us to study introspective learning in the first place. Therefore it is clear that we must set limits on the level of detail with which we attempt to model the system's reasoning, and we must develop guidelines for deciding what the appropriate level of detail is.

We approached the problem of determining what to assert about the reasoning from the opposite end, by considering what failures might occur, and working backwards to determine what assertions were necessary to detect them. Examining the reasoning process in detail to determine where failures might occur is a good general strategy for developing the kind of model of interest here. Because we want our model expectations to detect failures, it makes sense to start by determining what failures they are to detect. This approach proved beneficial to us in multiple ways. It emphasized the importance of implementation-specific assertions, as we will discuss in the next section. It also suggested the re-use of case retrieval for components of the planner itself, because of the similarity of failures and corresponding assertions in different components (as in the first failures given in Table 5 for the Indexer, Adaptor, and Executor).

Component	**Failures**	**Assertions**
Indexer	Failed to detect feature	The Indexer will use all applicable rules
	Added feature that was not there	The Indexer will never use a bad rule
Retriever	Incorrect index for retrieval	The Indexer will provide a proper index
	Failed to retrieve any case	The Retriever will find some match
	Retrieved an inapplicable case	The Retriever will retrieve applicable cases
Adaptor	Failed to apply some adaptation	The Adaptor will use all applicable strategies
	Could not completely adapt the case	The Adaptor has a strategy for every situation
	Made a bad adaptation	The Adaptor will improve the case
Executor	Failed to alter actions when necessary	The executor will use all applicable planlets
	Failed to reach goal	The robot will end at the goal
	Failed to complete plan in time	The robot will take N time-steps for each step
Storer	Stored a plan twice	The storer will recognize identical plans
	Stored plan under wrong index	A case is described by exactly one index

Table 5: Failure types

We developed a taxonomy of failure types for ROBBIE's planner to guide our choice of assertions, and also to suggest classes of possible repair for future extensions of the system. Table 5 shows a sample set of failure types and related assertions which were suggested by them, organized by the part of the system to which they refer.[1]

5.1.2 Level of detail for the assertions

One factor which distinguishes our work from others' is our view that a successful model for detecting failures and explaining failures must have assertions at multiple levels of detail. The model must include assertions which are abstract and describe the general flow of information and control through the reasoning process, and assertions which have specific knowledge of the implementation of the reasoning process. For

[1]Appendix B contains a full listing of our failure taxonomy.

detecting failures, and for specifying repairs, the model must contain implementation-specific knowledge. For explaining failures it is often necessary to trace the reasoning process across many stages of the reasoning process, spanning several components. Abstract assertions which describe the overall flow of control permit cross-component tracing to occur quickly, and can subsume with a single abstraction a collection of more specific assertions.

During the process of creating the failure type taxonomy, it became clear that many of the failures could only be recognized through implementation-specific assertions. How could a system determine that the adaptor "did not completely adapt the case" unless it could specify what a completely adapted case looked like? An abstract statement might not be sufficient to pinpoint the original problem, either: there might be more than one way in which the system could fail to completely adapt a case. Birnbaum et al. (1991) only suggest abstract assertions, but determining where they apply depends on more concrete knowledge.

5.1.3 Vocabulary for assertions

In the previous section we describe how we determined what assertions our model should contain, giving the assertions in English. In order to implement the model we require a vocabulary of primitives capable of capturing the meaning of our assertions. To keep the vocabulary as simple as possible, it is restricted to a limited number of primitives. Limiting the number of assertion types resists the temptation to include more assertions of lesser generality. Because the framework is to be general, it includes only general-purpose elements, and consists of terms independent of ROBBIE's own mechanisms. The vocabulary must also be flexible enough to represent assertions at abstract levels and detailed and implementation-specific levels.

Dependency:	**Structural:**
`depends-on-prev`	`has-value`
`depends-on-next`	`part-value`
`depends-on-spec`	`magnitude-value`
`depends-on-co-occurs`	`values-compare-by`
	`parts-compare-by`
	`magnitudes-compare-by`
	`form-of-structure`
	`member-of-structure`
	`has-type`
	`contains-part-of-type`
	`types-compare-by`

Table 6: Assertion vocabulary predicates

```
(magnitude-value retrieved-cases 1)
```

Figure 5.1: A typical assertion: *there will be one retrieved case*

The assertion vocabulary uses a simple first-order predicate calculus formulation for assertions, with a generic and restricted set of predicates. Table 6 lists the predicates available to describe assertions in our framework. Predicates in assertions must come from the list given, but *arguments* to predicates are not required to be generic. Instead, arguments refer directly to knowledge structures of the planner itself, to functions belonging to the planner and applied to knowledge structures, or to constant values. For example, the typical assertion shown in Figure 5.1 uses the generic predicate `magnitude-value`, but has an argument which refers to information from the planner (the list of retrieved cases) and an argument which is a constant value. The assertion says that the magnitude of the `retrieved-cases` structure will be one or, in other words, that exactly one case will be retrieved.

In determining what generic predicates are needed to describe the desired assertions, we found that abstract and implementation-specific assertions fell into different classes of assertions and used different elements of the vocabulary. Most abstract assertions depend on the values of their neighbors (predecessors, successors, or more specific children), rather than being directly evaluable in terms of information about the planner. Abstract assertions were described using the set of "depends-on" predicates. Specific assertions, on the other hand, naturally depend most directly on features of the underlying knowledge structures, and were described by assertion predicates specifying different kinds of features of the knowledge structures. We will consider these rough divisions of vocabulary types in more detail below.

Dependency assertions: Abstract assertions are most used for explaining failures, by focusing the search for other failed assertions. They organize information in the model, connecting different portions of the reasoning task, describing the flow of control in the system, and grouping together sets of specific assertions about the same reasoning task. The truth of such assertions depends on the truth of the assertions to which they are connected. Some assertions depend on their predecessors (such as the first Adaptor assertion in Figure 5.2), some on their successors, and others on the truth of some or all of their specifications. We developed the `depends-on` predicate types to describe these dependencies. Assertions using `depends-on` predicates direct the search for the root causes of failures by altering the search priorities. For example, when considering a `depends-on-prev` assertion, the search process will put off examination of all but the *predecessor* of the current assertion. Once the predecessor's success or failure has been determined, the same value may be assigned to the current assertion, and the relevance of other neighbors may be determined.

Features of underlying structures: Low-level assertions are most important in detecting failures and repairing them, as they have specific expectations about the

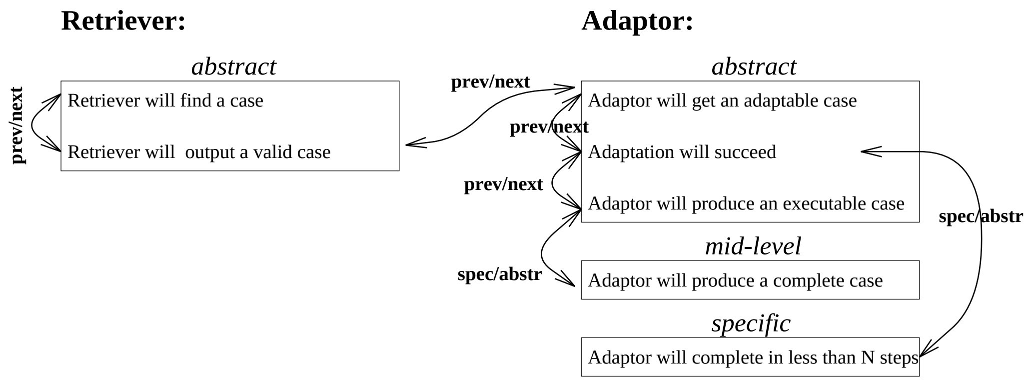

Figure 5.2: Sample assertions

implemented reasoning methods. Low-level assertions are more varied in type than their high-level counterparts; most assert some fact about an underlying knowledge structure. For instance, in Figure 5.2 the assertion "Adaptation will produce a complete case" is defined by features of the case produced which must be examined. We have developed a set of predicate types which correspond to different types of features of knowledge structures. Figure 6 includes elements of this set, including `magnitude-value` for examining the size of a structure, `part-value` for looking at features of a particular element within a structure, `form-of-structure` for considering the overall parts the structure contains, and `member-of-structure` which searches in a structure for a particular subpart.

5.2 Structure of the model

We described in the previous section what the assertions in the introspective model are like, but we must still describe how they are organized to form the model. The model is not simply a heap of assertions to be picked through at random; in order to function properly the assertions must be structured to provide information about how the parts of the reasoning process interrelate.

The model structure must conform to the goals listed at the beginning of the chapter: supporting failure-driven diagnosis, supporting detection and explanation of failures equally, allowing continuous monitoring for failures, and being general in form to allow re-use of the framework. Like the assertions in the previous section, the building blocks for the model structure should be generic, applicable across a wide range of modeled systems. Features of the structure of ROBBIE's underlying reasoning process should be reflected in the structure of the model, but not in the building blocks with which that structure is built.

The model structure must support both failure detection and explanation of failures. The first task requires a means to access the model in a sequential manner, focusing on directly verifiable expectations. The second task requires access to assertions across the model which are causally related to one another, regardless of where in the reasoning process they occur. A detected failure may have as its cause a flaw in reasoning that occured long before it in the overall reasoning process.

Figure 5.2, in the previous section, illustrates the structural features of the model. The modularity of the model is one important feature, in which assertions are clustered by the component of the underlying system to which they refer, and by how specific they are to the underlying system's implementation. This hierarchical structure and modularity maintain the generality of the model by permitting sections of the model to be retained while others are substituted when the underlying system changes. The hierarchical structure also makes the monitoring task more efficient by limiting the extent of the model which must be examined at a given point of reasoning. In addition to being modular and hierarchical, the model framework contains links from one assertion to another which denote causal relationships between assertions. Such links permit explanation of failures by permitting a search for related failures to cross cluster boundaries easily.

Figure 5.3 shows a sample cluster from the model, and one assertion within it (a larger segment of the model may be found in Appendix C). Each cluster has a name, and assertions within it may be identified by the cluster name combined with the personal name (in this case the number 1) which heads the assertion structure. Clusters which are abstract are linked through the `parts` slot to the more specific clusters for the same component. As this example is a specific cluster, there are no other parts. Assertions are represented as a sort of frame, starting with the assertion's local name. The next (unnamed) slot contains the description of the assertion itself, which in this case says that the Retriever will always select the closest matching case to return. The `when` slot describes the context in which this assertion is to be verified, it may contain the values `before`, `during`, `after`, or `on-failure`. The truth of the sample assertion is evaluated `on-failure`, when another failure has already been detected or there is other evidence that something has gone wrong. Next in the assertion structure is the list of knowledge structures on which this assertion depends. These describe portions of the planner's knowledge or the current reasoning trace; when these knowledge structures are not filled with relevant values, the assertion cannot be evaluated. The `links` slot contains the list of links from this assertion to others: in this case an abstraction link and a link to the next specific Retriever assertion. Finally, this assertion is associated with a class of repairs (not implemented) called `change-value`, indicating here that the weighting of features in retrieval should be altered if this assertion fails.

In the following sections we will discuss the structural features of the model in more detail, and describe how each addresses the goals for the model.

```
(retriev-specific
   (parts)
   (assertions
      (1 (forall (x in memory)
            (magnitudes-compare-by <=
               (closeness-goal x goals)
               (closeness-goal retrieved-case goals)))
         (when on-failure)
         (closeness-goal goals retrieved-case memory)
         (links (abstr (retriever 2))
                (next (retriev-specific 3)))
         (repair change-value))
     ...))
```

Figure 5.3: A typical cluster from the model, showing one assertion: *Every case in memory will be judged less (or equally) similar to the current problem than the case that was actually retrieved.*

5.2.1 Modularity

To facilitate our goal of developing a framework applicable to many systems, the model structure is highly modular, clustering assertions into small groups related by the component of the underlying system and their level of specificity. These modular constructs are based on system-independent building blocks: the central concepts are "component," "abstraction," and "sequence" which are not specific to any particular system. The modularity of the model permits the re-use of assertions where possible. For example, another case-based planning system could keep the more abstract clusters of the model intact, for each component similar to ROBBIE's, adding new lower-level details. A variation of ROBBIE which used a different adaptation mechanism could substitute new assertions for that component alone. This modularity also assists the monitoring task by making it easy to find only the currently relevant assertions. The monitoring phase need only consider the lower-level assertions associated with the current component of the reasoning task.

5.2.2 Hierarchical structure

The hierarchical structure of the model, as part of the overall modularity, permits re-use of portions of the model, as described above. It also separates different ways of thinking about the reasoning task: the abstract levels link components together and describe how information and control passes between them, low-level assertions describe portions of particular components and the specific information and algorithms they involve. The process of constructing a model of a system can then be broken down into development of an abstract-level description, followed by elaboration of each abstract cluster while making few changes to already-existing clusters. The hierarchical nature of the model plays a role in simplifying the monitoring task of the introspective reasoner, as we described above, but the existence of assertions at different levels of abstraction is also important for explaining failures. The abstract assertions permit explanation to skip quickly to causally-related but temporally-distant expectations, because they describe the process in general terms and connect whole portions of the reasoning process instead of individual steps. However, clustering abstract assertions separately from specific does not assist in explanation because the system cannot tell which assertions are causally relevant to a specific-level failure. Instead, the direct links between assertions in different clusters are used to pick out the relevant abstract assertions and to trace the reasoning process to other possible assertion failures.

5.2.3 Assertion to assertion links

Each assertion in the model is linked to the other assertions which are causally related to it. These links guide the introspective reasoner in explaining and repairing a detected failure by focusing on the most fruitful portions of the model. In other words, we can focus on those assertions which are most directly related causally to the detected failure by following links, instead of reasoning simply from the fact that two clusters of assertions occur at adjacent points of the reasoning process. Five different kinds of links distinguish different causal relationships between assertions. The links are used to guide the search for an explanation for a failure.

Two pairs of the five link types form symmetric relationships with each other. The first pair, `abstr` and `spec`, connect abstract assertions and specific assertions which are about the same portion of the reasoning process. The sample assertion in Figure 5.3 includes an `abstr` link to a higher-level assertion in a different cluster. A specific assertion usually has an `abstr` link to only one abstract assertion, but several specific assertions may be specifications of a single abstract assertion, which will have a `spec` link for each of them. The links `prev` and `next` relate expectations which precede or follow one another in the reasoning process, regardless of the cluster to which they belong. The example assertion contained a `next` link to the next in the retrieval process, but no `prev` links, since there are several points of entry into the retrieval process. The last link type is `co-occurs` which describes the relationship between two assertions which often succeed or fail together without having one of the aforementioned relationships.

Sequence links permit the system to trace back through the reasoning process to find where a failure first could have been introduced. Abstraction and specification links can help to control the search: moving from a specific failure to its abstract counterpart can permit a rapid move to consideration of other components of the system, and a high-level assertion that might be the root cause of a failure elsewhere may find a specification can verify it as the cause and suggest a repair.

5.3 Costs versus benefits

In combining introspective diagnosis and repair of reasoning failures with an existing system, we are adding yet another task, with associated costs, to the overall processing of the system. We must take into consideration what those costs may be, and how they may be offset by the benefits of performing introspective reasoning.

One cost of such a combined system is the additional time that must be spent doing the introspective reasoning, which may slow down the overall speed of the system. Another cost is the need to alter the underlying system to include communication with the introspective component in order to permit monitoring of the system in synch with its own reasoning. We must not forget the cost of constructing an introspective model, as well. Our framework for defining an introspective model is intended to assist in the model construction process, but building no model at all certainly takes less time.

We chose to focus on failure-driven introspective learning because it allows the system to focus on real opportunities to learn, and because it captures the kind of introspection seen in our everyday examples of people's reasoning. However, it also has the benefit of restricting the introspective task to a minimal overall effort. Other approaches to introspective reasoning, such as counter-factual reasoning (i.e., an introspective reasoner might consider what *would* have happened *if* the underlying reasoner had chosen differently) and reasoning about simulated situations (i.e., off-line, an introspective reasoner might consider many possible situation not actually faced by the underlying system), require much more reasoning effort, without the guarantee that there is any improvement to be done. By linking introspective effort to the existence of detectable failures, we ensure that introspective effort will intrude as little as possible into the performance of the system at its domain task.

By developing a simple and restricted vocabulary for communications between the underlying system and its introspective counterpart, we limited the kind of alterations to the underlying system which must be made to a reasonable set. To support this claim, adding the monitoring links to ROBBIE's own planner took less than two weeks, *once the model had been constructed*.

So far we have described the steps taken to minimized the costs of introspective reasoning, but there is another side to the story. The addition of introspective reasoning to an existing system may provide benefits which outweigh any additional costs incurred. An improvement in the system's reasoning mechanisms will lead to many improvements at the domain level; conversely, an unnoticed flaw in a system's behavior could affect the domain knowledge the system accumulates and the actions it chooses to take. Potential benefits include improved efficiency of the reasoning process, improved efficiency of problem *solutions*, and expanded problem-solving ability, solving problems that were previously impossible for the system.

As said in Chapter 1, complex systems demand adaptability from systems that operate in them. When the knowledge to be used is sufficiently rich, choices must be made about which features are currently relevant, a decision which may depend on other features of the current context, and which may change as dynamic elements of the domain alter without warning. Any feature forever excluded from consideration may turn out to be important under some set of circumstances. For instance, a designer might exclude the calendar date (for example, a holiday affecting traffic) from the knowledge of a delivery robot. However, if the robot has time constraints on it performance, it may fail to meet them on special holidays because of traffic, parades, or closed offices. The dynamic nature of a domain may create either obstacles or opportunities for the problem-solver: one day a snow bank blocks the way, another day a new sky-walk opens, permitting travel between two office buildings without going outside.

Introspective reasoning permits a system in a complex domain to adapt its reasoning methods when they prove inadequate to solve the given task, and to recover from gaps in its initial specification. It could alter its reasoning to include features it previously ignored, like the date or a current weather report. It could alter its reasoning to include new options, like checking the map of a given building for sky-walks to neighboring buildings.

In Chapter 7 we discuss the results of experiments testing the effects of introspective index learning on the performance of the ROBBIE system. We will show that introspective learning can improve, both the overall success and the efficiency of the altered reasoning process.

5.4 Summing up

In this chapter we described the introspective reasoning framework we developed for describing declarative models of ideal reasoning behavior, and detecting and diagnosing reasoning failures using such models. The model allows for failure-driven learning about the system's reasoning, supports both detection and explanation of failures equally, and is constructed out of generic pieces to permit re-use of the framework for different underlying systems.

The model contains assertions, some of which describe facts about specific pieces of the underlying reasoning, and some of which describe the reasoning process in more general terms. We claim that an introspective model must include knowledge about the reasoning process at multiple levels of abstraction; an abstract description of the reasoning process focusing on how control and information move around, tied to more and more specific details of how each portion of the task is implemented. Detection of failures, and repair of failures in the general case, requires specific assertions which can be easily examined to see if they are true of the current actual reasoning. However, explaining a failure requires tracing from the detected problem to the other portions of the model which may have affected it, and which may include previously undetected failures. Tracing through abstract assertions when assigning blame permits the system to rapid consideration of other assertions distant in terms of the reasoning trace but causally close in terms of the flow of control. To achieve both failure detection and failure explanation, therefore, the model must incorporate both abstract and specific levels of description.

Assertions in the model are clustered into groups which share the fact that they refer to a particular component of the reasoning process, and a particular level of specificity. When monitoring to detect failures, the introspective reasoner can quickly access only the most relevant assertions by selecting the single cluster containing specific assertions for the current component of the reasoning process. By itself, the modular, hierarchical aspect of the model is insufficient to explain failures: we also defined a set of causal links to connect each assertion to the other assertions which are directly related to it. By tracing the resulting graph, guided by the kind of link connecting two assertions, the introspective reasoner can quickly access other likely points of failure to determine the original point in the reasoning which is to blame for the detected failure. The model includes specification and abstraction links, links that indicate the sequence of reasoning, and causal links that connect assertions likely to fail or succeed together.

Ensuring that the introspective framework would be re-usable for other underlying systems was an important goal in determining how to represent the model. One result, with other advantages as described above, was the modular, hierarchical division of the model's assertions. Keeping assertions separated by component and specificity allows substitution of one cluster in the model for another to describe a new system. For example, another CBR system could use the high-level portions of ROBBIE's model and replace the underlying details.

The vocabulary with which to describe assertions includes a restricted and generic set of predicates which describe general classes of facts an introspective model might require. The restriction of the assertion vocabulary to these general predicates (with specific reasoning structures as arguments) will describe equally well features of other reasoning processes, while permitting us to describe the workings of the underlying reasoning process in sufficient detail for the task.

The building blocks out of which the actual model's structure is constructed are also designed for generality. They describe the underlying reasoning system and the model itself in terms which hold for most, if not all, reasoning systems: components, control, causal relationships, sequences, abstraction. The protocol for communicating between the underlying reasoning system and the introspective reasoner also restricts the information passed between underlying system and introspective reasoner to generalities, such as the reasoning component currently underway.

Application of introspective reasoning for the task of detecting and repairing reasoning failures of the system itself provides insights into the *general* knowledge requirements for other "reasoning about reasoning" tasks: predicting others' behavior, maintaining a dialogue, explaining behavior. The need for multiple levels of abstraction in knowledge of reasoning, and the ways in which that knowledge may interact (i.e., sequential versus causal relationships) may be generally true of other reasoning-modeling tasks.

In the next chapter we will discuss the implementation of this framework for the task of index refinement for ROBBIE. We will discuss the introspective index refinement task in detail, and describe what ROBBIE's model actually contains.

Chapter 6

Introspective Index Refinement in ROBBIE

> Our general framework for performing introspective learning has been applied to the specific task of index refinement: using introspective reasoning to detect when indexing criteria are inadequate and to determine new features to consider. This chapter describes the implementation of introspective index refinement.

In the previous chapter we described the framework we have developed for introspective reasoning. In this chapter we describe how the framework is applied to implement introspective index refinement. The processes for *monitoring* for failure detection and for *explaining* failures once detected, are general and apply to any potential failure or repair strategy. The *repair mechanisms* for ROBBIE, however, are specific to the problem of index refinement; other failures are detected and explained, but not repaired. Since the index problem is a central issue for case-based reasoning systems, this is a reasonable restriction at this time.

Selecting features to form an index for retrieving cases and judging the similarity of cases are well-known to be difficult problems for designers of case-based systems

(Kolodner, 1993a). Some features provided in an input problem description may be irrelevant, while other useful features may not be included explicitly in the input description. For example, in an input description of ROBBIE's planning goals, the robot's location `across` a sidewalk is often specified, but is irrelevant to the plans ROBBIE creates and remembers. A useful but unstated feature might be that a starting location and ending location are on the same side of the same street. Such a feature may be derived from the input description, along with many other possible features, only some of which will be relevant. It is not feasible to include all possible features, relevant or not, as explicit elements of either the input description or the resulting index. Even if all features could be included, their worth for guiding retrieval might vary enormously. For example, the weather seems a priori unimportant to the route planning task. Of course, even unlikely features could turn out to be important in *some* contexts: when making plans to travel around a beachfront resort, the weather can determine which roads are likely to be clogged.

We believe the best solution is to learn the features for indexing through experience. We have approached this learning problem through *introspective index refinement*: by using introspective reasoning to notice when and where existing features failed to produce correct retrieval results and altering the retrieval mechanism to include new relevant features. Index refinement is one application of introspective reasoning, but the framework we have developed could also be applied, in theory, to alterations at any point in the underlying reasoning process.

In this chapter, we first describe what the model of ROBBIE's planner contains, and how the model's organization is mapped from the organization of the planner itself. We then examine each phase in introspective reasoning for index refinement in turn, describing how it is implemented and providing sample output illustrating

ROBBIE's introspective processing.

6.1 Modeling ROBBIE

Before describing in detail how the introspective reasoner is implemented in ROBBIE, we will discuss the content of the model: what the model represents about ROBBIE itself. Each component of ROBBIE's planner corresponds in the model to a set of clusters of assertions about that portion of the reasoning process. Each cluster traces the steps of its component's reasoning process and describes how the component relates to others.

The model must provide expectations for the reasoning processes of each component of the planner: Indexer, Retriever, Adaptor, Executor, and Storer. Several different clusters contain assertions about a single component: these clusters of expectations vary in the level of specificity with which they describe the process. Each component is described by a cluster of abstract assertions: expectations about general stages of its reasoning process and its relation to other components. In addition, separate clusters describe the reasoning process for each component in increasingly specific terms, starting with abstract principles and ending with assertions which refer to actual knowledge structures and implementation details of the component in question.

To take one component as an example, the portion of the model for the Retriever consists of two clusters of assertions: one abstract and one describing implementation details. The abstract cluster describes the retrieval process for ROBBIE as follows: the retriever gets a valid index from the Indexer, its similarity assessment ranks cases correctly, it succeeds in examining cases in memory and selecting the best one(s), it returns the right cases, and they are valid. The specific cluster contains assertions which fill in the details: for example, what it means for cases to be ranked correctly, how the retrieval process selects cases, and how many and what kind of cases should be returned for a given situation.

The retriever is connected to the components whose processing precedes and follows it in the case-based planning task (Indexer and Adaptor), but it is also connected to the components of which it is a sub-part: components implemented using case-based retrieval (Indexer again, Adaptor, and Executor). These connections are mirrored in the model by links between assertions in the Retriever's clusters and related assertions in other clusters for other components. The links make explicit the causal relationships between the Retriever and the rest of the system.

6.2 Monitoring

The introspective reasoning system depends on the underlying planner to indicate when monitoring may take place. The planner sends a signal to the introspective reasoner describing the current state of the reasoning process. The planner signals at frequent, regular intervals in its normal processing, but may also signal when it has discovered an unexpected problem on its own. In that case the message to the introspective reasoner will include a description of the discovered problem. Table 7 describes a few sample messages the planner can produce and what they tell the introspective reasoner.

Under normal circumstances, the monitoring phase of the introspective reasoner is used to verify that the underlying reasoning is performing as expected from the model of ideal processing. The assumption is made that the reasoning process is

`(executor start)`	Routine monitoring of the Executor before it starts
`(adaptor new-strat steps-taken)`	Routine monitoring of Adaptor between strategy selections
`(executor no-planlets failure)`	A failure has appeared in the Executor, no planlets matched the current situation

Table 7: Monitoring messages from the planner

performing perfectly unless ROBBIE determines that an assertion has failed. ROBBIE's introspective reasoner uses the information in the monitoring message to select the relevant portion of the model for verification. It first selects the most specific cluster of assertions for the current component of reasoning, then selects from that cluster the assertions relevant to the current point of the component's processing. For simplicity and generality only three points of a given component's process are distinguished: `before`, `during`, and `after`. In addition, a fourth relevance class is denoted by `on-failure`, which describes assertions which are relevant only when some other failure has already been detected. An `on-failure` assertion is often also restricted to relevance either `before`, `during`, or `after` the current component's processing. Distinguishing these relevance classes focuses the monitoring effort on a small number of assertions.

Figure 6.1 illustrates how assertions may relate to a given component's processing. Assertions which are described as `before` their component refer to the state of the reasoning process when control is passing to the current component, `during` assertions refer to any point in the middle of the component's task, and `after` assertions refer to the state when the component's processing is complete and control is passing to the next component.

The following sample run illustrates introspective monitoring during the early

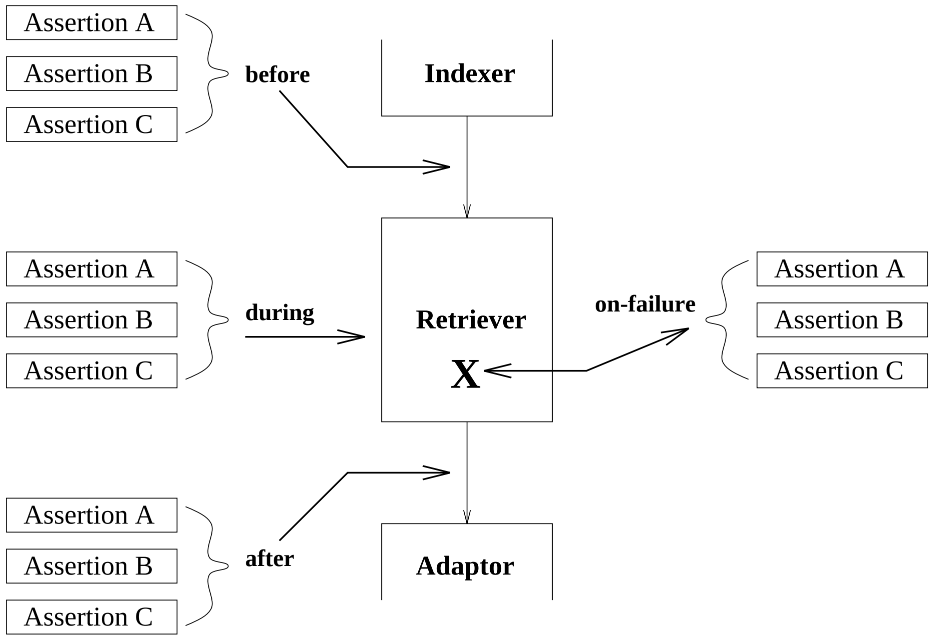

Figure 6.1: Assertions refer to points in processing: `before`, `during`, `after`, and `on-failure`.

phases of the planning process: before and after the Indexer creates a `plan` index.[1] Monitoring at the start of the Indexer's process will only evaluate assertions that are labeled as `before`, and not labeled as `on-failure`. At the end of the Indexer's process, those assertions labeled with `after` (and not `on-failure`) will be evaluated. Other assertions are "skipped" since they are not currently relevant.

```
Rob(CBR)>> No current goal; please input next destination: ind1
Rob(CBR)>> New destination is:
The north side of birch, the 100 block, 1 feet along

Rob(MBR)== MONITORING: (indexer start)

Rob(MBR)== Checking out indexer, before
   Rob(MBR)== Skipping indexer-specific assertion:        "On-failure" assertion
       (values-compare-by find-features final-solution context)
```

[1]Under normal circumstances, output describing introspective monitoring is suppressed, but may be enabled by setting ROBBIE into the correct output mode.

```
   Rob(MBR)== Checking indexer-specific assertion:
       (has-type goal-loc location?)
   Rob(MBR)== Assertion checked out
   Rob(MBR)== Checking indexer-specific assertion:
       (has-type current-loc location?)
   Rob(MBR)== Assertion checked out
   Rob(MBR)== Skipping indexer-specific assertion:      "After" assertion
       (contains-part-of-type index from-part location?)
   Rob(MBR)== Skipping indexer-specific assertion:      "After" assertion
       (contains-part-of-type index to-part location?)
   Rob(MBR)== Skipping indexer-specific assertion:      "After" assertion
       (contains-part-of-type index other-part spec-index?)
   Rob(MBR)== All indexer-specific assertions considered
Rob(MBR)== No assertion failures found
...
Rob(Ind)>> Index is:
...
Rob(MBR)== MONITORING: (indexer end)

Rob(MBR)== Checking out indexer, after
   Rob(MBR)== Skipping indexer-specific assertion:      "On-failure" assertion
       (values-compare-by find-features final-solution context)
   Rob(MBR)== Skipping indexer-specific assertion:      "Before" assertion
       (has-type goal location?)
   Rob(MBR)== Skipping indexer-specific assertion:      "Before" assertion
       (has-type current-loc location?)
   Rob(MBR)== Checking indexer-specific assertion:      "After" assertions
       (contains-part-of-type index from-part location?)  now relevant
   Rob(MBR)== Assertion checked out
   Rob(MBR)== Checking indexer-specific assertion:
       (contains-part-of-type index to-part location?)
   Rob(MBR)== Assertion checked out
   Rob(MBR)== Checking indexer-specific assertion:
       (contains-part-of-type index other-part spec-index?)
   Rob(MBR)== Assertion checked out
   Rob(MBR)== All indexer-specific assertions considered
Rob(MBR)== No assertion failures found
```

A special case for the monitoring phase occurs when the planner itself has detected a problem with its own processing and informs the introspective reasoner that a failure must have occurred. In this case the monitoring phase assumes that a problem exists, and considers `on-failure` assertions as well as the ordinarily relevant assertions. As an example of this kind of assertion, the model assumes that the indexing criteria include all relevant features so long as the best case is retrieved with them. An assertion in the model states that the Indexer uses all relevant features,[2] but it is classified as an `on-failure` assertion and thus is only examined to see if any features are missing when an indexing or retrieval failure has been detected.

Planner notices a catastrophic failure, signals the introspective reasoner for help. Relevance is automatically set to "on-failure."

```
Rob(Exe)>> Problem, can't replan yet!!!
Rob(CBR)>> Failure discovered; executor is lost!
Rob(MBR)== MONITORING: (executor replanning failure)

Rob(MBR)== Checking out executor, on-failure
   Rob(MBR)== Checking exec-specific assertion:
...
```

Relevance is now on-failure

Once the relevant assertions have been selected, each assertion is passed to a simple interpreter which determines if the assertion can be evaluated given the current situation and, if so, evaluates it. A relevant assertion may depend on information (i.e., from the planner's knowledge structures) which is not yet available. For instance, during the adaptation process supplemental knowledge structures are constructed, but parts of the structures may be incomplete at any given time when introspective monitoring occurs. Assertions related to those structures may *appear* relevant even when the information upon which they depend is absent. If information is unavailable to evaluate an assertion, the assertion will be stored in a list of "suspended" assertions

[2]This assertion appears in the previous sample output and is skipped.

until the introspective session ends or the information becomes available. Assertions are also suspended during the explanation phase of introspective reasoning.

If an assertion can be evaluated at the current time, the interpreter does so, storing the result and the assertion on a list of "completed" assertions. The "completed" list ensures that an assertion with a given context will only be evaluated once in an introspective session (monitoring, explanation, and repair phases). In addition, as one assertion's value may depend on another's, the "completed" list serves as a reference for determining other assertions' values.

If all the relevant assertions for monitoring prove to be true of the current state of the reasoning process, the introspective reasoner sends an `okay` message back to the planner, and ends its operation until the next monitoring message from the planner. If one assertion is not true of the current reasoning process, it is added to a list of failed assertions while the rest of the relevant assertions are evaluated. In this way, multiple failures appearing at the same time will all be noticed. The list of failures is then passed to the explanation phase of the process.

In this sample run, a failure is detected during the Storer's operation: an unretrieved case has a more similar solution than the retrieved case:

```
Time = 15:03:00

Rob(Sto)>> Reconstructing finished plan
Rob(MBR)== MONITORING: (storer start)

Rob(MBR)== Checking out storer, before
   Rob(MBR)== Checking storer-specific assertion:
      There exists x in (retrieve-by-solution final-solution),
      such that the rule
      (values-compare-by equal? retrieved-case x) holds
```

The assertion says that the retrieved case has the most

```
   similar solution to the final solution. Checking this assertion
   requires a retrieval from memory, hence recursive monitoring of
   Indexer and Retriever:
   Rob(MBR)== MONITORING: (indexer start)
Rob(Ind)>> Creating plan solution index
...
Rob(MBR)== Assertion in storer-specific failed:
   There exists x in (retrieve-by-solution final-solution),
   such that the rule
   (values-compare-by equal? retrieved-case x) holds
Rob(MBR)== All storer-specific assertions considered

Rob(MBR)== Assertion failures found; explaining failures
...
```

6.3 Explaining failures

The explanation of a detected failure involves searching through the assertions in the model for other failures which are causally related to the detected failure. The goal is to find the earliest related assertion failure in the reasoning process, the likely source of the detected failure. The explanation should lead the system to a repair strategy to correct the failure for the future. If more than one failure is detected at the same time, the explanation process is repeated for each failure, but work once done is not repeated: by storing assertions which have been evaluated, along with the context in which they were evaluated and the resulting truth value, the introspective reasoner only has to check each assertion in a given context once.

Each assertion has associated with it links to other causally related assertions. In terms of these links, the model can be seen as a graph with an assertion at each node. This graph is searched using a modified depth-first search starting at the detected assertion failure.

From a given assertion, the search proceeds to each linked neighbor in a depth-first fashion, where the order in which the next neighbor is selected depends on the type of link to the neighbor and the predicates forming the current assertion itself. Be default the search considers first `abstr` links, then `prev`, then `spec`, and lastly `next` links. This ensures that the search will head generally backwards in the reasoning trace, and will move toward more abstract levels first. Moving quickly to the abstract level places consideration of other components to be a high priority in the search. Some assertions, like `depends-on` assertions, alter the order in which their neighbors are considered to place priority on their particular requirements: a `depends-on-next` assertion will extend the search first by following its `next` links, and then by following the rest of its links in a normal order.

```
Rob(MBR)== Assertion failures found; explaining failures

Rob(MBR)== Checking links for (storer-specific 2) failure      Original failure
Rob(MBR)== Checking link (abstr (storer 1) storer-specific 2)  Abstraction
Rob(MBR)== Checking storer assertion:
    (depend-on-spec the storer is given a good final solution)
Rob(MBR)== Assertion depends on children
Rob(MBR)== Assertion in storer failed:       Fails because depends on failed child
    (depend-on-spec the storer is given a good final solution)
Rob(MBR)== Found related assertion failures in (storer 1)

Rob(MBR)== Checking link (prev (executor 5) storer 1)     Predecessor of parent
Rob(MBR)== Checking executor assertion:
    (depend-on-next the executor will produce a good plan)
Rob(MBR)== Assertion depends on next assertion
Rob(MBR)== Assertion in executor failed:     Fails because depends on next
    (depend-on-next the executor will produce a good plan)
Rob(MBR)== Found related assertion failures in (executor 5)
...
```

If a given assertion depends on information which is not available, either from the planner's knowledge structures or from another assertion, evaluation of it will be suspended and the assertion stored until the information becomes available. As new information becomes available to the explanation phase, the list of currently suspended assertions is reconsidered and any which are now ready to be evaluated are removed from the list and evaluated. Assertions which include `depends-on` predicates are suspended if the assertions upon which they depend are unevaluated, but such unevaluated assertions are placed at the front of the search queue for immediate consideration. An assertion which forces the addition of linked assertions to the queue causes those links to be removed from the queue once the values of the linked assertions are no longer needed.

A "depends-on" assertion must be suspended until its children are evaluated.

```
...
Rob(MBR)== Checking link (prev (executor 4) executor 5)
Rob(MBR)== Checking executor assertion:
    (depend-on-spec the executor will end up at the goal location)
Rob(MBR)== Assertion depends on children
```

The executor assertion must be suspended until its specifications have been evaluated:

```
Rob(MBR)== Suspending assertion...
Rob(MBR)== Have suspended assertion, now adding links
...
```

The search ends when no further assertions remain in the queue to be considered: either every assertion related to the original one has been evaluated or it has been determined to be unnecessary. More than one failure might be found, suggesting competing repairs. Since we have implemented a single repair strategy, deciding between suggested repairs is not a problem, but must be considered in a comprehensive repair system. In general, ROBBIE will prefer earlier failures, and those at the end of a chain of failures rather than in the middle.

It is possible for the search queue to become empty without a root cause failure or a potential repair being found. This is especially likely if information from the planner is missing and many assertions are suspended. If no further progress can be made in explaining a detected failure, the introspective reasoner suspends the entire explanation process, stores its current status, and waits for a later point in the planner's reasoning progress to continue the explanation effort. Later opportunities for introspective monitoring will automatically re-start the explanation process.

Explanation phase must be suspended for lack of information.
Later on it is restarted.

```
   Rob(MBR)== All assertions suspended, no root cause found
Rob(MBR)== Suspending explanation phase
Rob(MBR)== Returning ‘‘watch-out’’ warning to planner
...
   Rob(MBR)== MONITORING: (executor new-planlet)
   Rob(MBR)== Restarting explanation phase
...
```

6.4 Repairing failures

A description of the failures found, including suggested repairs, is passed to the repair module from the explanation module. ROBBIE currently ignores any suggested repairs other than index refinement, and simply signals the planner that a failure was found which could not be repaired.

```
(1 (values-compare-by find-features final-solution context)
   (when on-failure)
   (find-features final-solution context)
   (links (abstr (indexer 1))
          (next (indexer-specific 2)))
   (repair add-case))
```

Figure 6.2: Indexer `on-failure` assertion for finding features

6.4.1 Finding missing features

During the explanation phase, if a failure is traced back to the Indexer, scrutiny will fall on an assertion which states that the Indexer includes all the relevant features of a situation in its indexing criteria (See Figure 6.2). To avoid unnecessary overhead, this assertion is normally not checked: the current indexing criteria are assumed correct unless they have proven to be flawed. Limiting the search for new features to known retrieval failures ensures that the indexing criteria remain as small as possible. In determining the truth of this assertion for the current situation, the final solution and the un-retrieved best match are examined for features which they share, but which are not already explicitly included in the indexing criteria.

ROBBIE considers a feature a candidate for inclusion in the indexing criteria if it is present in both the final solution and its best match. In addition, it must be a feature derivable from the original index for the plan, since it must be possible to detect it in the original input description before a route plan has been constructed.

To evaluate whether a feature should be added to the indexing criteria, the planner's procedure `find-features` is applied to the two relevant cases. This procedure distills the basic indices of the two relevant cases into a more abstract form, and applies a set of heuristics to determine which of a pre-defined set of feature types are present in both indices. If any feature type can be instantiated for both cases, and its instantiation is not already in use in the indexing criteria, then the resulting

`both-same`	Plan starts and ends at given locations
`start-same`	Plan starts at a given location
`end-same`	Plan ends at a given location
`start-same-x`	Plan starts at a given longitude east (same north/south street)
`start-same-y`	Plan starts at a given latitude north (same east/west street)
`end-same-x`	Plan ends at a given longitude east (same north/south street)
`end-same-y`	Plan ends at a given latitude north (same east/west street)
`stay-on-x`	Plan's longitude doesn't change (stays on same north/south street)
`stay-on-y`	Plan's latitude doesn't change (stays on same east/west street)

Table 8: Feature types

instantiated feature is a candidate for addition to the indexing criteria.

The basic indices are simply the starting and ending locations for each case; they are converted into a rough Cartesian coordinate system (accurate to within about 10 units) and then the coordinates are compared. Cartesian coordinates are used in this case to avoid difficulties stemming from the different possible representations of a single location. A location described as "99 units along the 100 block of the west side Oak Street" is actually the same as "the southwest corner of Oak and Birch:" the similarity may be hidden by the normal location description but becomes clear when distilled to numbers describing how far east and how far north the location is.

ROBBIE's feature detector mechanism currently utilizes a set of detectable feature types from which it selects new criteria. Table 8 lists the feature types included in the detector process. For each feature type, ROBBIE can determine from the distilled descriptions of the two cases (four locations in Cartesian coordinates) whether the two cases share an instantiation of that *type* of feature. If both cases can instantiate

```
(start-same-x
  ,(lambda (fr1 to1 fr2 to2)
     (~= (x-val fr1) (x-val fr2)))
  (indexer plan ?frx ?y1 ?x ?y2)
  5)
```

Figure 6.3: A typical feature type rule

a feature type the same way, then a new candidate feature is created by substituting in the values which are shared in common.

Figure 6.3 shows the structure which describes the `start-same-x` feature type. The structure contains a simple procedure for determining whether the feature type applies, the prototype of an index for the *indexing* rule which would be created to implement an instance of the feature type, and an *importance* value which rates the feature type's importance against other possible feature types. The function determines whether the two cases do "start at the same longitude value." It says that `start-same-x` is a feature type that applies to the two cases if the X-values of the starting locations are roughly equal. From this general feature type, a specific feature could be created for a particular X value, and an indexing rule added which searches for that feature.

The feature that is learned must apply to individual cases, since it is to be added to the indices of individual problems. Therefore, an instantiated feature for the `start-same-x` feature type will contain a particular starting X value. For example, if the two cases could instantiate the feature type `start-same-x` with the same longitude value of "250", then the feature created would be "starts at longitude value 250", and the indices of other cases which start at that longitude value be annotated with that specific feature. Feature types are instantiated into actual features with varying degrees of specificity. For example, the feature type `stay-on-x` will only be instantiated once, since it does not refer to any particular latitude or longitude values.

The feature type `both-same` will be instantiated to a particular starting and ending location, and will apply only to plans that start and end at roughly (though not exactly) the same locations. More general features, such as `stay-on-x`, are preferred over more specific ones.

The feature detector attempts to instantiate each feature type with the current two cases. If none match, then it is assumed that the original assertion is true: the Indexer really did use all relevant features in building an index for retrieval.[3] If feature types are successfully instantiated for the current two cases, then the assertion is false, and the instantiated feature with the highest importance value is selected as a suggested root repair. This assertion failure, and the feature that caused it, are added to the list of "completed" assertions, just as any other assertion in the explanation phase would be, and are passed to the repair phase when the time comes.

```
Rob(MBR)== Checking link (prev (indexer-specific 1) indexer-specific 2)
Rob(MBR)== Checking indexer-specific assertion:
   (values-compare-by find-features final-solution context)
Rob(MBR)== Unretrieved plan is: g930
Rob(MBR)== Final Solution is: g935
Rob(MBR)== Searching for correlations with heuristics
Rob(MBR)== Came up with: (end-same-x)       One feature type matches
Rob(MBR)== Correlation built is:
(end-same-x
  (indexer plan ?x ?y1 ?tox ?y2)
  (*spec-index* end-same-x 936))
 In the resulting indexing rule (see next output trace) ?x, ?y1, and
 ?y2 will remain as variables to be unified with a current situation's index
 later on. ?tox, however, refers to the X-value of the goal locations of
 the two cases, and will be filled in with a specific number by the repair
 component.
Rob(MBR)== Assertion in indexer-specific failed:
```

[3] We recognize that features exist which are not part of this hierarchy; future work could extend this feature detection mechanism to a more general approach, e.g., involving explanation-based generalization (Mitchell et al., 1986; DeJong & Mooney, 1986).

```
   (values-compare-by find-features final-solution context)
...
```

In order for index refinement to be suggested as a repair strategy, the explanation phase must have traced the failure to faulty indexing *and* discovered a feature in the initial description of its problem which was not considered explicitly and which would have led to the correct retrieval. Therefore, the index-refining repair strategy receives from the explanation phase all the information it needs to effect the repair. Changing the indexing criteria to include the new feature requires first building an indexing rule to be stored in memory and retrieved by the Indexer when applicable. This will ensure that the new feature is appended to every applicable index that appears in the future. In addition, the indices of every case in memory, and of the current case (which may or may not be stored in memory already) must be altered to include the new feature, if they implicitly contain it. Therefore the repair strategy applies the new rule it creates to the index of the current situation, and to every case in memory.

Having found a repairable failure and finished considering other possible failures, the results are passed to the repair phase:

```
Rob(MBR)== Completed search for deeper cause of (storer-specific 2) failure
Rob(MBR)== No repair known for failure:
   There exists x in (retrieve-by-solution final-solution),   Another related
   such that the rule                                         failure with no
   (values-compare-by equal? retrieved-case x) holds          known repair
Rob(MBR)== Repairing (storer-specific 2) failure from assertion:
   (values-compare-by find-features final-solution context)  Root failure
Rob(MBR)== The general repair specified is add-case
Rob(MBR)== The rule created is: end-same-x                   New rule for
(indexerg937                                                  memory, with
  (indexer plan ?x ?y1 40 ?y2)                                ending X-value
```

```
  (add-to-key end-same-x (*spec-index* end-same-x 936)))     instantiated
Rob(MBR)== Rule added
```

Once a repair is complete, the planner receives a `fail-fixed` *message and continues*

```
Rob(Sto)>> Storing plan in memory under name: g935
Rob(MBR)== MONITORING: (storer end)

Rob(MBR)== Checking out storer, after
   Rob(MBR)== Skipping storer-specific assertion:
      There exists x in (retrieve-by-solution final-solution),
      such that the rule
      (values-compare-by equal? retrieved-case x) holds
   Rob(MBR)== All storer-specific assertions considered
```

Once the repair phase is completed, successfully or not, the introspective reasoner informs the planner of the result of its introspective processing. Table 9 lists the results the introspective reasoning system might send to the planner. Possible outcomes include no problem being detected, a problem being detected and successfully repaired, or variations of un-repairable problems. The planner may choose to alter its behavior on the basis of these introspective outcomes. ROBBIE's planner usually gives up if it receives a `watch-out` or `fail-in-progress` response. For example, if the introspective reasoner determines that the Adaptor is not making progress towards a completely adapted plan, the planner will choose to stop trying to adapt the current case. A more powerful planner might respond differently to the last two messages: by expending differing amounts of extra effort checking its progress and being prepared to re-plan.

`okay`	No assertion failures found
`fail-fixed`	Failure found and fixed, continue
`fail-in-progress`	Failure found, explanation suspended, continue with caution
`watch-out`	Failure found and explained, no repair, continue with extreme caution

Table 9: Messages passed from introspective reasoner to planner

6.5 Summing up

ROBBIE's introspective reasoning mechanisms, for monitoring the reasoning process for failures and explaining detected failures, implement the model-based introspective framework we described in Chapter 5. The model itself describes the entire reasoning process of ROBBIE's planner, both abstractly and with implementation details. Expectation failures which occur at any point in ROBBIE's reasoning process are detected, and often explained. ROBBIE, however, focuses on one kind of repair: refining the set of features used in constructing indices for plan cases.

The introspective reasoner's ability to suspend evaluation of assertions when the information on which they depend is unavailable permits the model to contain assertions whose values depend on other assertions. This sort of dependency may connect the value of an abstract assertion to its specific counterparts which actually examines the planner's knowledge structures and performance. Suspending assertions also permits the introspective reasoner to accommodate the changing information at the planning level: if the planner has not yet created some knowledge structure, the introspective reasoner can simply set aside those assertions which depend upon it.

Dividing assertions into clusters makes determining relevant assertions easier, as does including notations in the assertion structure that describe at what point of its component's process a given assertion is valid.

The explanation phase must potentially examine a large number of assertions in the model in determining which assertions are responsible for a detected failure. By automatically guiding the search toward earlier components of processing and permitting assertions themselves to alter the order in which their neighbors are evaluated, the assertions which are examined may be minimized.

ROBBIE contains the framework of a general repair phase, but only one repair strategy has been studied. This strategy — a particularly important one for case-based reasoning — extends the features used in the indexing criteria, and depends on the discovery during the explanation phase of a feature that exists in both the final solution and in the best match case which is not used in the current indexing criteria. The addition of such a feature alters the assessment of similarity of the two cases and prevents similar faulty retrievals in the future. The repair strategy must create a new indexing rule for the discovered feature, add it to memory for retrieval by the Indexer, and apply the rule to the plan cases already in memory.

The feature detection mechanisms are currently simple, and restricted to a set of feature types defined to capture common similarities between cases. Despite this simplicity, features are learned introspectively which do capture important regularities in the cases ROBBIE experiences, and which do permit its performance to improve when new features are learned introspectively. In Chapter 7 we discuss experiments which show the effectiveness of ROBBIE's introspective learning process.

Chapter 7

Experimental Results

> Empirical evaluation provides more convincing proof of a system's performance than examining individual examples. We describe experiments which show that ROBBIE performs better with both case and introspective learning, compared to case learning alone. We also examine the effect of problem order on ROBBIE's performance. We describe circumstances in which introspective index refinement does and does not improve performance.

Many artificial intelligence systems are evaluated solely by studying a few selected examples believed to typify the problems of interest, in order to verify that the system performs as expected. This approach may include a detailed analysis of the system's methods and its most important features. It raises the question, however, of the breadth of the system's good performance: the suspicion may lurk that the examples studied are the only examples for which the system works properly (McDermott, 1981), or that this performance is not typical over the long term (Minton, 1990). This has led to a call for systematic evaluations of artificial intelligence systems (e.g., (Cohen & Howe, 1988)).

Our previous chapters have discussed the *in principle* benefits of introspective learning. This chapter presents empirical tests of our system. To our knowledge, these are the first empirical tests of any system using introspective reasoning to learn from reasoning failures.

This chapter examines the effects of case learning and introspective learning on ROBBIE's performance. To gauge the full effect of both kinds of learning we must view the performance over time and the presentation of many goals. A single example of case-based learning will inform us that the system can store a new case correctly, but the real question is whether the new case can be retrieved and applied to improve the system's later planning. Similarly, from a single example of introspective index refinement we can learn whether the system finds and adds a feature which is currently appropriate, but to determine the ultimate effect of including that feature in the indexing criteria we must examine the long-term performance.

Because of the need to gauge the long-term effects of case-based and, especially, introspective learning, and our desire to explore the full extent of ROBBIE's potential, we designed a set of large-scale empirical tests. These experiments examine ROBBIE's performance over long sequences of goals. In this way, cases and indexing features learned early on have a chance to affect the later processing of the system. Different task sequences provided a range of different levels of difficulty for the system. Some contained goals which were extensions of previous problems, making ROBBIE's task easier. Others were random collections of goals, so that a given problem might bear little similarity to any problem ROBBIE had seen before.

The empirical tests we designed focused on two questions:

Benefit of introspective index refinement: The first, and most important, question was to compare ROBBIE's performance under a given sequence of goals using only case learning by the planner, versus its performance on the same sequence using both case learning and introspective index refinement. A difference between the performances would demonstrate the effect of adding introspective reasoning to the system. We predicted that introspective index refinement would improve the performance of the system no matter what the properties of the goal sequence. New features, determined introspectively when the system knows that the current features are not correct, improve the planner's information about how to select a case for retrieval, and hence should allow the more appropriate cases to stand out in future retrievals, eliminating less appropriate cases.

Effect of problem order: The second question addressed by these experiments is the effect on the learning and performance of ROBBIE of the order in which goals are presented to it. Problem order is well-known to strongly affect learning in general, and learning of CBR systems in particular (e.g., (Bain, 1986; Redmond, 1992)). The question of how to select a sequence of situations to present to a system has also been examined in reference to the training of connectionist networks (e.g., (Cottrell & Tsung, 1989; Elman, 1991)). problem order to affect the *introspective learning* of ROBBIE by altering the pool of cases in memory. With different cases in memory, the closest case in memory to a given solution may differ, eliciting different features to be added to the indexing criteria. The effect of problem order is especially acute for ROBBIE because its initial knowledge of routes in its domain is limited.

We compared ROBBIE's performance across different sequences to see how the difficulty of the problem order affected performance: some sequences were designed to facilitate learning, others to obstruct it. Sequences which facilitated learning were called "well-ordered."

In order to create a well-ordered sequence of goal locations, we first needed to define what we meant by "well-ordered." Redmond (1992) suggests problems presented early in a case-based learner's experiences should cover the range of possible situations, to lay a foundation for later learning. We claim that in addition each problem should push the boundaries of the system's capabilities without extending so far that the problem is impossible for the system to solve. A sequence of goals which gradually increases in complexity and distance from the original cases in memory should maximize the effectiveness of the case-based reasoner's learning.

In this chapter we describe the experiments performed on ROBBIE: how the experiments were designed, what measures of ROBBIE's performance were used, and what results were found. We discuss what we conclude from these tests.

7.1 Experimental design

Before presentation of each sequence of goals, ROBBIE and the world simulator were set to the same initial conditions. The world map used is shown in Figure 7.1, the same map in which much of the development of the ROBBIE system took place. ROBBIE's case memory initially contained three plans, all of which applied to locations in the southwest corner of the map. The small size of the memory, and the fact that the plans in it did not span the entire map, stack the deck against ROBBIE and allow us to examine how a learner with limited initial information could expand its knowledge to encompass its domain.

The initial indexing criteria for plans contain just the starting and ending locations for a given problem. The introspective learning that occurs during the test run, presentation of the sequence of goals to ROBBIE with a particular initial condition,

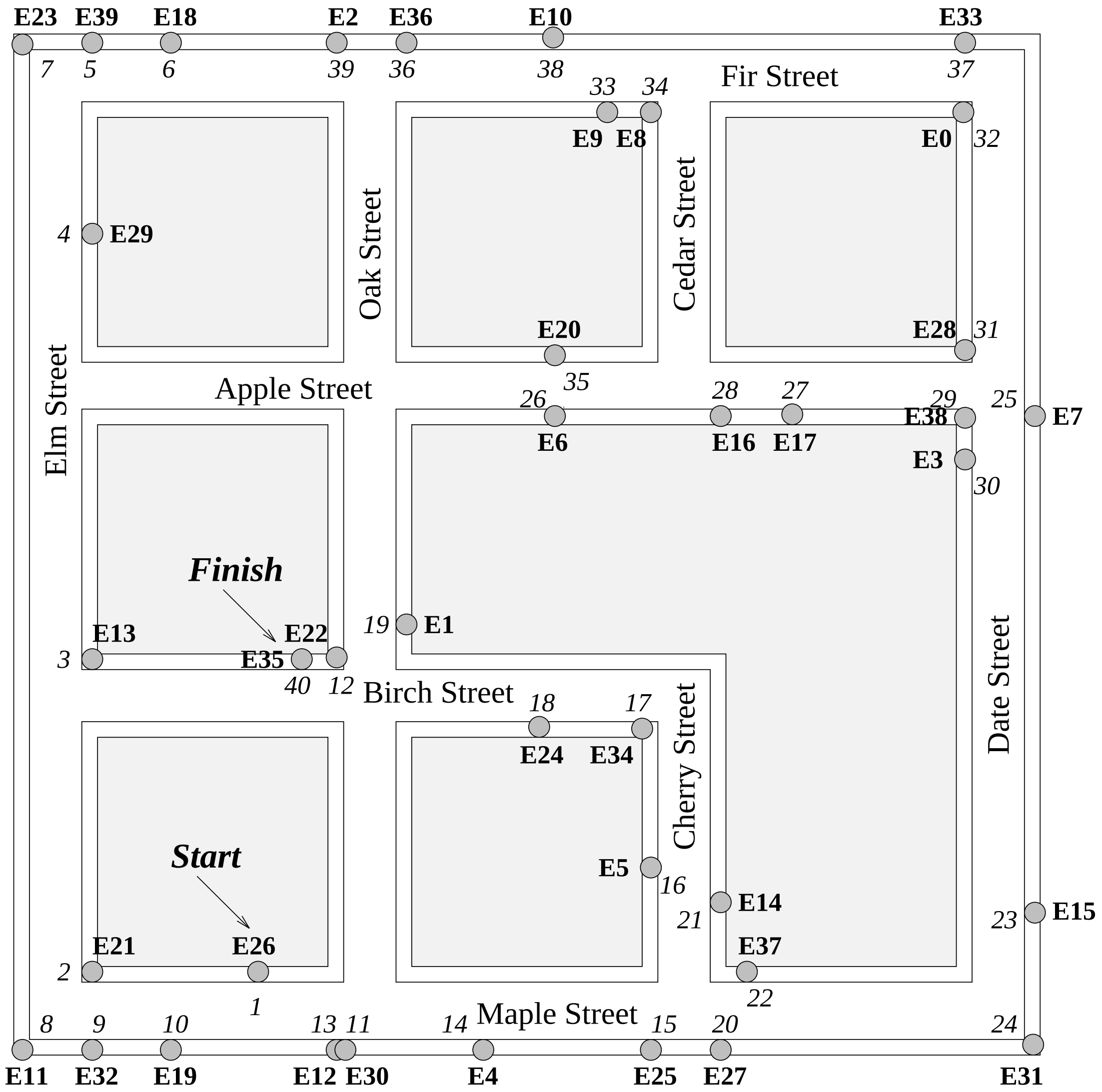

Figure 7.1: Map for experiments

provides the only opportunity for index refinement. In half the test runs, the learning ability of the introspective reasoner was disabled, so that only domain learning by storing new cases could occur.

If we selected locations by hand to use as goals in the sequences, we would risk some unconscious bias in the locations. Therefore, we generated 40 locations at random (the locations are shown in Figure 7.1). Each sequence of goals is a list of these 40 locations, in some ordering. After solving 40 problems under good conditions, ROBBIE usually has a significantly diverse case memory, and has had several (i.e., 5-10) opportunities to apply introspective learning.

In these experiments, ROBBIE's initial location was set to the first location in the sequence, and the next in the sequence was taken to be the first goal location. ROBBIE considered each location in turn, planned from its current location to that goal, executed the plan, and went on directly to the next goal location in the list. For the purposes of these tests, when ROBBIE failed to reach a goal and was unable to recover, ROBBIE's location was set to the failed goal and the sequence continued from that point.

A total of 26 sequences were created to test ROBBIE, ranging between "well-ordered" and random in their arrangement of locations. One sequence was generated by hand to be very well-ordered; the other twenty-five formed five groups generated by altering the original sequence. The principles defining when a sequence is "well-ordered," outlined above, guided our creation of the handmade sequence. The sequence was designed to extend the case base a little bit at a time: it started with goal locations similar to those in the case base, and gradually moved to goals further away from the initial cases. The italicized numbers in Figure 7.1 indicate the order of locations in the handmade sequence, which is listed in Figure 7.2.

Position:	1	2	3	4	5	6	7	8	9	10
Location:	E26	E21	E13	E29	E39	E18	E23	E11	E32	E19
	11	12	13	14	15	16	17	18	19	20
	E30	E22	E12	E4	E25	E5	E34	E24	E1	E27
	21	22	23	24	25	26	27	28	29	30
	E14	E37	E15	E31	E7	E6	E17	E16	E38	E3
	31	32	33	34	35	36	37	38	39	40
	E28	E0	E9	E8	E20	E36	E33	E10	E2	E35

Figure 7.2: Sequence of locations in handmade sequence

Each of the five groups of sequences contained five sequences generated in the same way. Four of the five groups were generated by permuting the original sequence to different extents. A permutation of a sequence involves selecting two positions in the sequence at random (i.e., the third and seventeenth positions) and swapping the locations at those positions. Each group's sequences were formed by permuting the original the same number of times. The first group had 10 permutations (corresponding to about 25% of its elements being swapped), the second 20 permutations, the third 30 permutations, and the fourth 40 permutations. The fifth and final group contained sequences which were purely random orderings of the 40 locations. As it turned out, even when 40 permutations of the original had been done, a few locations remained in their original positions; including purely random sequences ensured no residual bias from the well-ordered original would exist.

For each sequence ROBBIE was tested an equal number of times with both case and introspective learning, and with case learning alone. Each initial configuration was run multiple times because ROBBIE chooses at random between two candidate cases which appear equally similar. It is often the case, however, that the two cases produce different results which may affect later processing: one might fail or cause a new feature to be learned. To factor out idiosyncrasies of any particular run, we ran each sequence twenty times with introspective index refinement and twenty times without it. Altogether, ROBBIE was presented with a total of 41,600 goals: 26 sequences, each 40 goals long; each sequence was run 20 times with introspection and 20 without.

7.2 Performance measures

The performance of ROBBIE for a given test run was measured in two ways. First, we counted the number of plans which were successfully completed and added to memory by the end of a test run, and second, we counted the number of cases considered good matches by the Retriever for each given retrieval. The number of successfully learned plans was used to estimate the overall success rate of the system: by comparing the number of successes between two test runs we could rate ROBBIE's task performance in each case. We predicted that the success rate should be higher when using introspective learning of indexing features, since learned features could improve the focus of the retrieval process to retrieve an appropriate case more often. The number of good matches for each retrieval was used to evaluate the efficiency of the retrieval process: fewer cases considered good matches indicates that the Retriever is focusing on the best matches and ignoring other more marginal matches. As index refinement focuses the retrieval process, the number of cases considered should drop. In sum, we expected introspective re-indexing to increase the success of ROBBIE *at the same time* as it decreased the work done in considering cases for retrieval.[1]

ROBBIE only learns a new plan case when it has successfully executed the plan:

[1] It would also be possible for introspective learning to cause retrieval to become too restrictive, decreasing success; verifying that this is not *commonly* the case is another reason for empirical tests.

when it fails to reach the goal no case is added to memory. Thus, we can measure the success rate by counting the number of plan cases in memory after a test run. This count includes the three original cases, but their inclusion affects nothing as it is the same for every test run. We evaluated the statistical significance of differences between test runs with case learning alone and runs with case and introspective learning, using a technique called "bootstrapping" to extrapolate what the normal distribution of differences would be if the addition of introspective learning had no effect (the null hypothesis). We then performed a t-test on our *observed* differences, using the mean and standard deviation of the bootstrapped population to compute the significance of the observed differences. If the observed difference is sufficiently distant from the population mean of the null hypothesis, ("sufficiently distant" is defined by the value of t), then it is highly unlikely that the observed difference could be due to random chance. It is therefore extremely likely that an actual difference in performance exists.

While the measure of ROBBIE's success rate examines the state of the case memory at the end of a test run, the measure of the Retriever's efficiency is continuous across a test run. Each time a plan retrieval was requested, we stored the percentage of plans in memory which were considered good matches. We could then plot the percentage considered against the sequence of retrievals for a given test run to gauge the effect of new cases and new indexing criteria on the later retrieval process. Because of the small number of plans in memory initially, the early retrieval percentages are high, but the percentages become more reliable as the case memory grows. We found, however, that the percentage considered a good match varied widely within a test run depending on the problem presented. Whenever a situation relatively dissimilar to all previously seen problems was presented, the percentage of memory considered a good match was very high. A better sense for the performance of the retrieval process over time was gained by examining the *successive averages* of the sequence of retrievals done during a test run. For each retrieval performed during a test run, we calculated the average of its percentage considered good matches and all previous percentages in the sequence. We then plotted these successive averages against their position in the retrieval sequence to get a sense of the *trend* of retrieval percentages.

We kept track of other measure of ROBBIE's performance, although we did not use them directly in our evaluation. We noted where failures occurred, what the failure was, and kept track of the total number of failures for a test run. We also noted where re-retrieval was applied, and whether it succeeded or not. We examined where introspective index refinement took place, and did some analysis of its effects, as we describe below. These additional measures and more qualitative analyses verified that our expectations about ROBBIE's processes and learning were correct.

7.3 Effects of introspective learning

In this section we describe the results of our experiments in comparing ROBBIE's performance with case learning alone to its performance with both case and introspective learning. From an informal examination of the mean of the cases learned for the runs of a given sequence, we found that, on average, test runs with introspective learning outperformed test runs with case learning alone. In fact, the performance with introspective learning showed an improvement for all but one run. We performed statistical analysis of the difference between success rates with and without introspective learning for each sequence, and found that 21 of the observed differences were highly significant, while 5 showed equivalent performance with and without index refinement. The one test run that informally showed poorer performance with introspective learning proved to be statistically equivalent. We suggest that the five sequence producing anomalous are due to the potential effect of index refinement to over-restrict the retrieval process.

7.3.1 Success rate

We measured the success rate as the number of cases in memory at the end of a test run. Each initial configuration of the system was run twenty times to gauge the different possible outcomes due to random decisions during the planning process. We examined the frequency of each success rate and compared frequencies for test runs with introspective index refinement and for those without. Figure 7.3 shows the frequencies of different success rates for one of the random sequences as a histogram (actually two histograms displayed together). The dark bars represent the number of times each success rate occurred for the test runs with index refinement; the lighter bars represent the frequency of that success rate for test runs using only case learning. The distribution of outcomes is not strictly a bell curve, but the rough peak of the test runs with only case learning are between 13 and 15, while the peak of the test runs with both case and introspective learning is higher, between 15 and 17. It is clear from the figure that case learning alone tends to produce lower success rates than when case learning is combined with introspective learning.

By averaging the twenty success rates, we can compare the performance with and without index refinement for each sequence. We will do so informally first, then discuss statistical evaluation in Section 7.3.2. Figure 7.4 shows a breakdown of the results. The averages for a particular sequence with and without index refinement

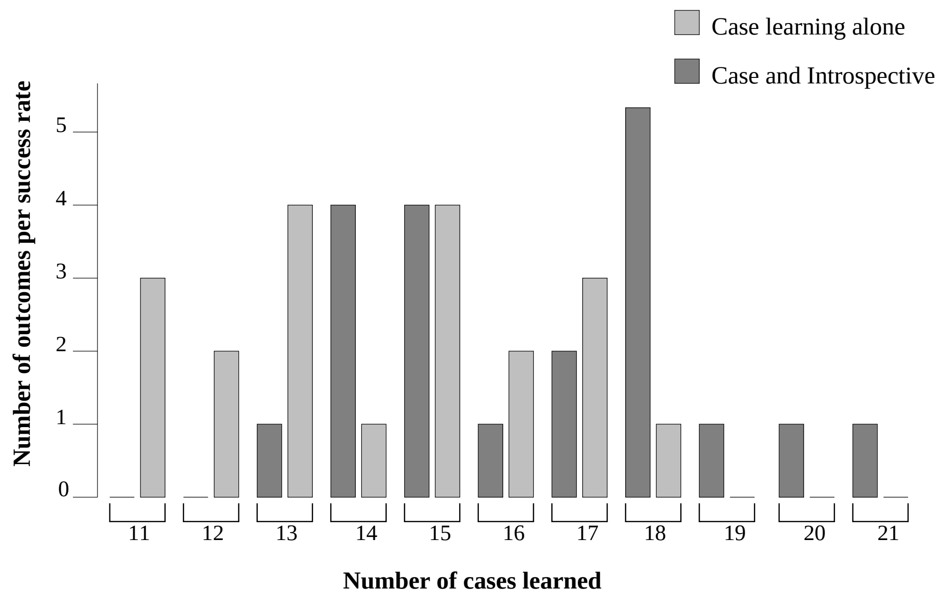

Figure 7.3: Histogram of success rate frequencies for sequence Random 3 shows a lower range of values for case learning alone and a higher range for both case and introspective learning

are side by side with the darker shaded bars indicating the presence of introspective index refinement and the lighter shaded, case learning alone. For all but one sequence, ROBBIE with introspective learning succeeded more often, on average, than ROBBIE with only case learning, as we expected.[2] For each group of sequences equally permuted from the original, the mean success rate varied widely. In addition, the size of the difference between runs with introspective reasoning and those with case learning alone varied a great deal. In general a large difference between performance with introspective learning and without it correlated with a higher average success rate, and a greater improvement in retrieval efficiency.

[2]The fact that the anomaly is the last sequence is pure chance.

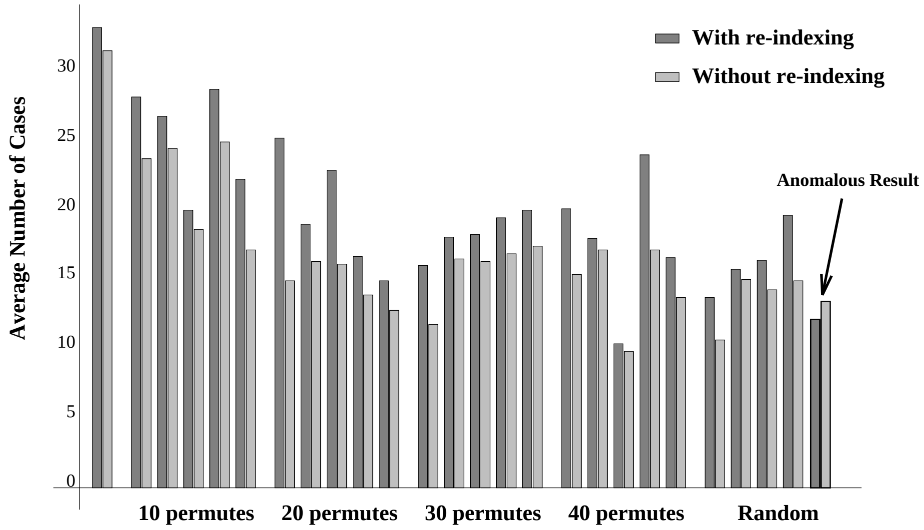

Figure 7.4: Average success rates for each sequence, dark bars with index refinement, lighter bars without it: runs with index refinement outperform runs without it

7.3.2 Statistical significance of differences

In order to determine just how significant the differences we saw in Section 7.3.1 were, we evaluated the mean difference for a given sequence between test runs with index refinement and test runs with case learning alone[3]. With a larger set of test runs, or a more well-behaved distribution of success rates, we could perform a standard "t-test" on the mean differences to determine their statistical significance. However, examination of the success rate distributions (like the histogram shown in Figure 7.3 above) for particular sequences showed that not all sequences had normal distributions (an assumption of the standard t-test). We performed instead a variation of the t-test which makes no assumptions about the distribution of success rates in our samples.

We applied a statistical method called *boot-strapping* (Efron & Tibshirani, 1993) to produce a "population" of mean differences to which to compare the observed mean difference of a sequence. We wished to determine how likely it would be that the

[3]I.e., take the difference of the average success rate with index refinement and the average success rate with case learning alone.

observed mean difference of the success rates of a sequence would arise if, in reality, the addition of introspective learning had no effect on ROBBIE's performance (the null hypothesis). The lower the probability of the mean difference arising at random, the higher the probability that the difference we observed was significant. The elements of the population of mean differences, therefore, should represent the null hypothesis, mean differences which draw equally from outcomes with introspective learning and without it. This distribution naturally centers around zero.

To create a population to which to compare the actual mean difference, bootstrapping extrapolates from the individual outcomes (success rates, in this case). For a given sequence, all the outcomes (both with and without introspective learning) are placed in a single pool. From that pool, two samples the same size as the original samples are selected by picking elements at random without removing them from the pool. The mean difference of these two randomly selected sets is calculated and stored. This process, repeated thousands of times, produces a normal distribution of differences under the null hypothesis for a given sequence.

For our experiment, each randomly-selected sample contained twenty elements chosen randomly from a single pool of the success rates for all forty test runs of a sequence (both with and without index refinement). For each sequence, we collected 50,000 re-sampled mean differences. Figure 7.5 shows the resulting distribution of mean differences for a typical sequence (Permuted 20c). On the graph we have marked where our observed mean difference falls; it is clearly in the distant wings of the bell curve, which indicates a strongly significant difference. Nevertheless, we must still determine the actual probability of that difference occurring under the null hypothesis, using a variation of the t-test.

Because we generated the distribution of mean differences for each sequence under

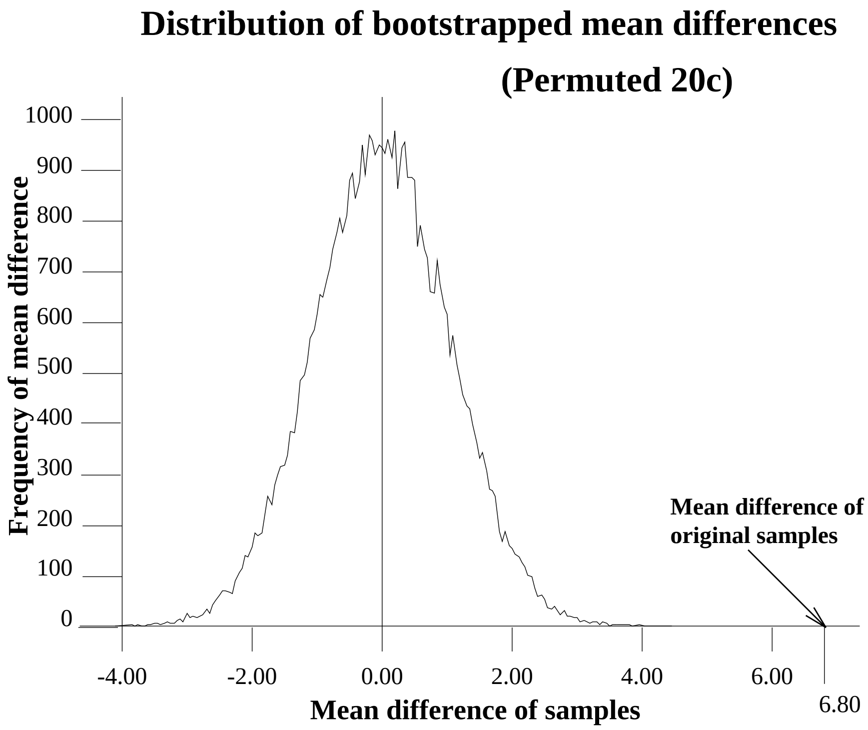

Figure 7.5: Population distribution of mean differences created by bootstrapping for Permuted 20c; the population represents the null hypothesis, the distance of the observed mean from the curve suggests it may be significant

the null hypothesis, we could calculate the *standard error of the difference between means* (σ_{diff}) as the standard deviation for that population distribution. We could then calculate t under the following formula:

$$t = \frac{M_{ci} - M_c}{\sigma_{\text{diff}}}$$

Where M_{ci} is the mean of the success rates of the sequence with both case and introspective learning, and M_c is the mean of the success rates of the sequence with case learning alone. Using that formula we found the values for t given in Table 10.

From t and a measure of the degrees of freedom of this problem (the sum of the sample sizes, minus 2, or roughly 40) we could determine the probability that any observed difference occurred by chance, i.e., that case learning alone performs just as well as case and introspective learning. The probability values are shown in the fifth column of Table 10. The lower the value of p, the higher the likelihood that our observed difference is real and significant. We consider any value of p less than or equal to 0.05 (5%) to indicate a significant difference. For most of the sequences, p was far less than one-tenth of one percent, indicating that the difference was highly significant. As the sequences became less well-ordered, they tended to have lower values for p, closer to the borderline for significance.

Five sequences shown in bold lettering had observed differences which proved statistically insignificant. For those five sequences, introspective learning performed just about the same as case learning alone. We discuss in a later section why introspective learning might not produce an improvement, but it is useful to note that while ROBBIE with introspective learning did not perform measurably better than ROBBIE with case learning alone on those sequences, it did not perform measurably *worse*

Sequence	Sample Mean Difference	Bootstrapped Stand. Dev. σ_{diff}	t Value	Probability Difference is Insignificant	How Signif. (***)
Handmade	1.75	0.288	6.067	$p < 0.001$	***
10Perm.(a)	4.30	0.521	8.259	$p < 0.001$	***
10Perm.(b)	2.45	0.320	7.662	$p < 0.001$	***
10Perm.(c)	1.50	0.273	5.492	$p < 0.001$	***
10Perm.(d)	3.90	0.526	7.420	$p < 0.001$	***
10Perm.(e)	5.05	0.679	7.440	$p < 0.001$	***
20Perm.(a)	10.25	1.383	7.413	$p < 0.001$	***
20Perm.(b)	2.75	0.634	4.338	$p < 0.001$	***
20Perm.(c)	6.80	1.044	6.516	$p < 0.001$	***
20Perm.(d)	2.80	0.406	6.897	$p < 0.001$	***
20Perm.(e)	2.10	0.450	4.663	$p < 0.001$	***
30Perm.(a)	4.35	0.379	11.473	$p < 0.001$	***
30Perm.(b)	1.35	0.871	1.550	$p < 0.1$	
30Perm.(c)	2.20	0.593	3.710	$p < 0.001$	***
30Perm.(d)	2.70	1.164	2.321	$p < 0.025$	*
30Perm.(e)	2.60	1.098	2.368	$p < 0.025$	*
40Perm.(a)	4.70	0.646	7.272	$p < 0.001$	***
40Perm.(b)	0.85	0.394	2.160	$p < 0.025$	*
40Perm.(c)	0.65	0.780	0.834	$p < 0.25$	
40Perm.(d)	6.75	0.917	7.361	$p < 0.001$	***
40Perm.(e)	2.95	1.773	1.664	$p < 0.25$	
Random (1)	3.00	0.759	3.952	$p < 0.001$	***
Random (2)	0.65	0.515	1.261	$p < 0.25$	
Random (3)	2.25	0.696	3.231	$p < 0.005$	**
Random (4)	4.80	1.303	3.683	$p < 0.001$	***
Random (5)	-0.30	0.720	0.417	$p < 0.4$	

Table 10: Statistical significance of observed mean differences for each sequence: significant differences are marked with *, **, or *** (indicating level of significance), insignificant differences are unmarked

either. The anomalous sequence noted in Section 7.3.1 turns out, under statistical analysis, not to have significantly worse performance with index refinement than it does without: the performance is statistically equivalent.

The fact that the likelihood of equal performance increases as the "orderliness" of problems presented decreases is another indication that problem order deserves careful consideration, especially with a system as initially knowledge-poor as ROBBIE.

7.3.3 Improved retrieval efficiency

The second comparison between case learning alone and case and introspective learning together examines the extent to which learning new indices improves the efficiency of ROBBIE's plan retrieval process. We measured this by considering the percentage of stored plans ROBBIE collected as good matches in the first phase of its retrieval process. We examined how retrieval changed over the course of a sequence of goals by plotting the percentage of cases in memory considered good matches against the sequence of retrievals. The sequence of retrievals made during a test run does not match up exactly to the sequence of goal locations, because each application of the re-retrieval process results in a total of three cases retrieved from memory (the original unadaptable case plus two cases for the subgoals) instead of the usual one case. The fact that different test runs might use different numbers of retrievals for a given goal location also meant it was difficult to generate an average percentage considered for each goal location. However, by putting runs of a sequence together on one set of axes, we could see commonalities between runs in terms of the percentage of cases that ROBBIE considered similar. Figure 7.6 is an example of this sort of graph.

That index refinement has a marked effect on the retrieval process can be seen in Figure 7.6, which shows a single test run of the original well-ordered sequence using

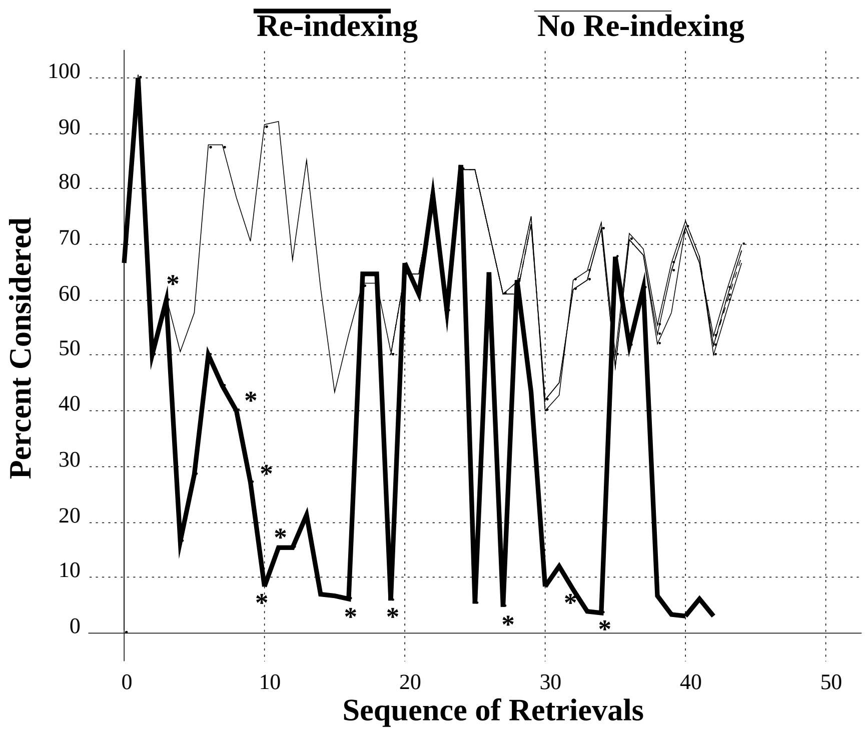

Figure 7.6: Percentage of good matches for test runs of the "well-ordered" sequence: dark line is one test run with introspective re-indexing, stars show where re-indexing took place, light lines are five runs without re-indexing. Re-indexing causes much lower percentages on some problems

index refinement as well as case learning (the heavy line) compared to five runs of the same sequence with case learning alone (the light lines). Each point of the test run where introspective learning took place is marked with an asterisk. The runs without index refinement are almost identical in the percentages considered at each point. They appear almost as a single line, varying only due to an occasional re-retrieval that shifts where a particular goal occurs in the sequence of retrievals. The percentages for the introspective test run follow the same course at the beginning of the sequence, but the percentage of good matches drops sharply after the first instance of introspective learning. While the percentage of good matches for the introspective run does occasionally increase at later points to nearly the level of the runs without index refinement, it repeatedly returns to a much lower level.

It is interesting to note that after the first instances of introspective learning, most of the later introspective learning take place at points where the percentage of cases considered good matches is low, instead of at high-percentage points. This suggests that at the low points, retrieval might have had a narrower focus than needed, and the new features learned should cross boundaries formed by previous features to cluster cases differently in the future. The lack of introspective learning at high-percentage points also suggests that when no cases in memory are similar to a goal, considering a larger percentage of cases during retrieval is appropriate. With a much larger case memory, the percentage would obviously be much smaller even when a new problem was relatively dissimilar.

As Figure 7.6 demonstrates, the percentage considered good matches fluctuates between very high and very low levels from one retrieval to another. This fluctuation made it difficult to analyze the *trend* of retrievals directly from the graph of percentages. We gained a better perspective by plotting the “successive averages”

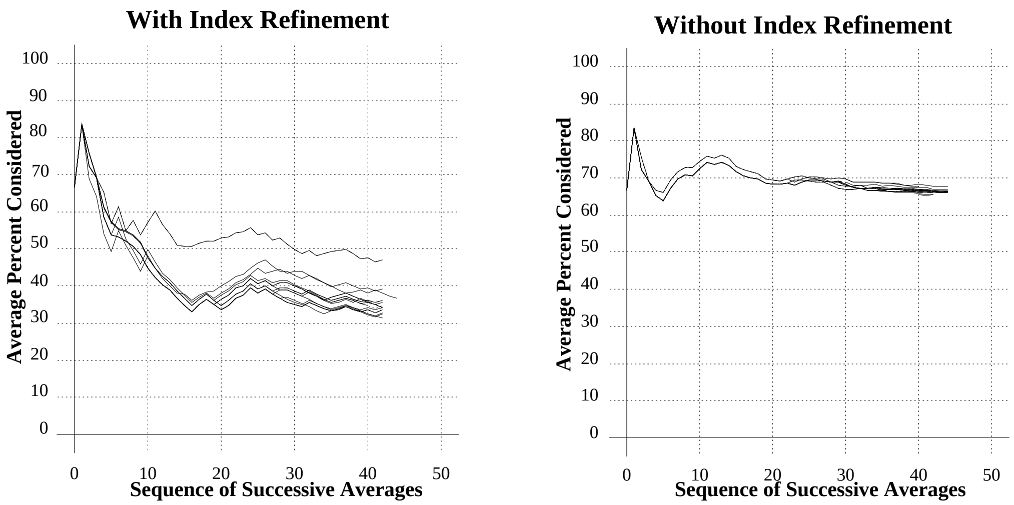

Figure 7.7: Successive averages for the handmade case: a large decrease in retrieval percentages for introspective runs compared to non-introspective runs

of the percentage considered at each point in the sequence of retrievals. Figure 7.7 shows the resulting graphs for a sequence which demonstrated one of the most dramatic changes in the average percentage. Figure 7.8 shows the same kind of graph for a sequence with moderate change to the percentages. The corresponding graph for the worst anomalous sequence is in Section 7.5 below; it shows almost no difference between the performance with and without introspective index refinement.

For the sequence shown, the twenty runs with introspective learning are shown on the left and the twenty without it on the right. The average percentages considered with index refinement declined over time to a much lower level than those without: for the best-performing sequence, almost all runs dropped below 40% when introspective learning was permitted compared to 65-70% without it; for the typical sequence, the percentage ranged between 30-45% instead of 60-70%. A similar pattern of decline appeared in every sequence except the anomalous one. The decrease was most dramatically better (both in terms of how low the percentage became and

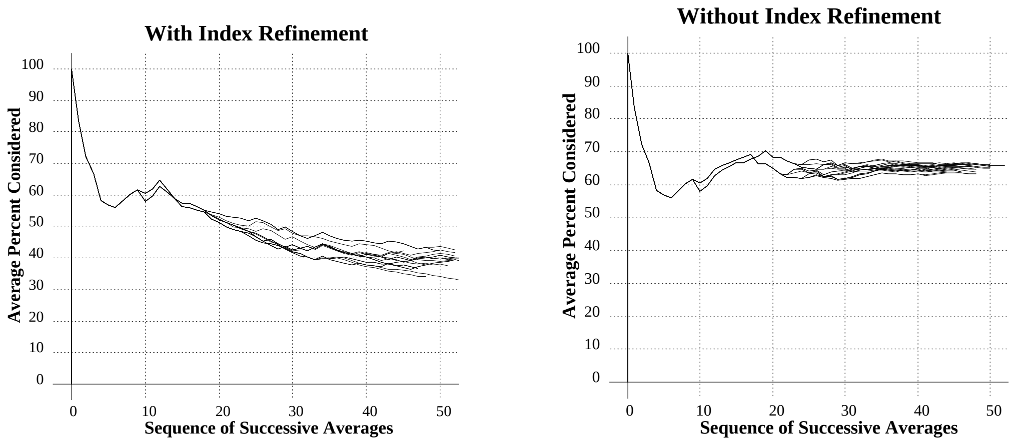

Figure 7.8: Successive averages for a typical sequence: typical difference between introspective and non-introspective graphs

in how consistently runs displayed that low percentage) for those sequences which also demonstrated higher overall success rates and greater *differences* between the success rates with and without introspective learning.

7.4 Effects of problem order

The sequences used to test ROBBIE's performance were designed to vary in the extent to which each assisted ROBBIE by building gradually from ROBBIE's initial knowledge. By using sequences that varied so widely, we expected to elicit a general sense for ROBBIE's performance across a variety of situations. We were also interested in comparing ROBBIE's performance on sequences with different degrees of "orderliness" in order to examine the effects of problem order on both the case-based and introspective portions of ROBBIE. We expect that ROBBIE's performance should degrade as the sequences of problems become more random. Test runs with introspective learning should degrade more slowly with disorder, because introspective learning should make the most out of the cases that are in memory. However, the quality of new features added to the indexing criteria depends on the quality of cases in memory. When the sequence of goals is random ROBBIE's planner will perform the most poorly: the chances that a given problem will be similar to a known one are smaller compared to a well-ordered sequence. If the planner succeeds less often, introspective learning will also suffer because the case size will remain small and be less diverse.

To focus on the effects of problem order, we examined the success rates for the different groups of sequences with equal amounts of disorder to them. We calculated the success rate for each group by averaging the average success rates for each member of the group. The resulting averages for test runs using only case learning are plotted on the left in Figure 7.9. The success rate is high for the sequence designed by hand to meet our criteria for being "well-ordered:" out of 40 new problems the sequence averaged only 10 failures. As the sequences are permuted to differ from the well-ordered sequence the performance drops off rapidly. After 20 permutations have been done on the original sequence, the average performance has dropped to a low level that is essentially equivalent to the more random sequences, failing on 25 to 30 out of 40 goals.

A similar pattern of decline appears when we consider the averaged success rates for groups of sequences using both case and introspective learning, shown on the right in Figure 7.9. However, ROBBIE with both case and introspective learning is more resistant to the effect of poorly ordered problem sequences than ROBBIE with case learning alone. The average runs with introspective learning degrade to the random sequence level more slowly, showing that the inclusion of introspective learning does mitigate the negative effects of problem order.

We did not perform a statistical analysis of the effects of problem order due to

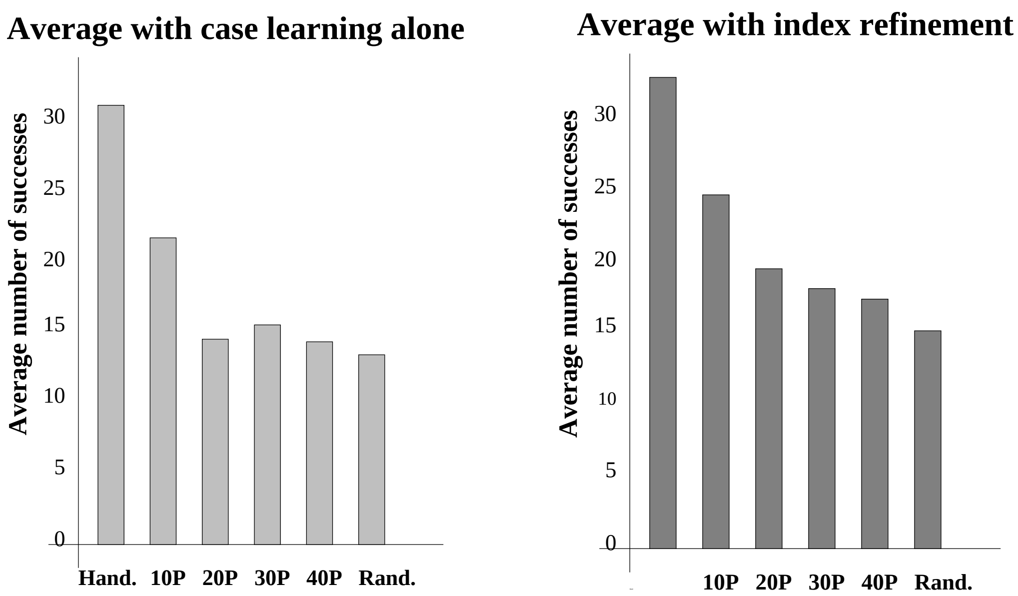

Figure 7.9: Average success rate of groups of sequences: with case learning the average drops with random problem orders, with introspective learning the average declines more slowly

the complexity of comparing sets of sequences with bootstrapping techniques. For now, we are content with an informal analysis showing that performance drops as expected, and that introspective learning can mitigate, but not overcome, the effect. A rigorous formal analysis similar to that done for the effects of introspective learning remains for future work.

7.5 Anomalous sequences

Five sequences showed performance with introspective learning that was too similar to the performance of the sequence with case learning alone to be judged statistically significant. One such sequence had a negative mean difference between its performance with and without index refinement (i.e., from looking at the averages, introspective learning seemed worse). We can group the five sequence by the probability value (p)

of each, which estimates the likelihood of seeing the observed mean difference when in fact the real performance is equal. The anomalous sequence, Random 5, had the highest probability that there was no real difference: between 25% and 40%. Two sequences were in the next percentage range, 10% to 25%: the sequences Permuted 40e and Random 2. The final two sequences nearly showed a statistically significant difference, having a likelihood of no difference between 5% and 10%.

The performance of these anomalous sequences may derive from the interaction of a difficult sequence of goals, ROBBIE's small initial memory, and index refinement which added irrelevant or unnecessary features to the indexing criteria. Introspective learning of new indexing features only benefits the planner if those features apply to a useful subset of the situations the planner will see in the future. The most compelling explanation for the anomalous sequences is that introspective learning simply did not create new features which would always be useful in later situations.

Because ROBBIE's re-indexing depends on having a case in memory which shares important features with the current solution, the smaller the case memory, the less likely a good match will be found and a useful feature learned. Therefore, when the planner repeatedly fails at goals, as in a difficult random sequence, few cases will be added to memory, and index refinement may select features which are not widely applicable. These features will focus the retrieval process on fewer cases but, because the features may not cluster cases in useful ways, the retrieval process may occasionally ignore a case which should have been considered, leading to a failure in planning. If the failure prevents ROBBIE from achieving that goal location, it may not be traced back to a missing indexing feature, since no "final solution" exists to which to compare cases in memory. Therefore, a better case to retrieve may not be discovered and the correct features may not be learned.

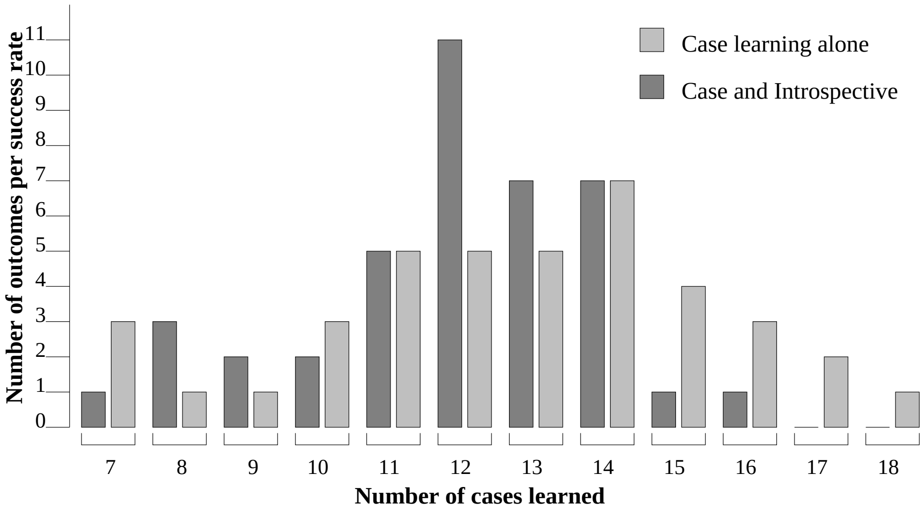

Figure 7.10: Histogram of success rate frequencies for anomalous sequence: the peaks for case versus case and introspective learning are close together (12-14), case learning alone extends higher

The anomalous sequence Random 5 strongly fits the profile for over-restriction by index refinement described above. Test runs of the sequence using case learning alone show a very low success rate compared to all other sequences, random or permuted. Figure 7.10 shows the frequency of the range of success rates for test runs of the anomalous sequence, both with and without introspective index refinement. As for the previous histogram, the lighter bars show values for case learning alone, darker bars show values when introspective learning is enabled. The most common success rates lie between the 10 and 15 range, meaning about 10 out of 40 goals were successfully achieved. The range of success rates for this anomalous sequence is quite wide compared to other sequences.

The tendency of index refinement, when combined with poor case learning performance, to over-restrict the retrieval process by learning irrelevant features is evidenced by the measure of retrieval efficiency. Plotting the test runs for the sequence

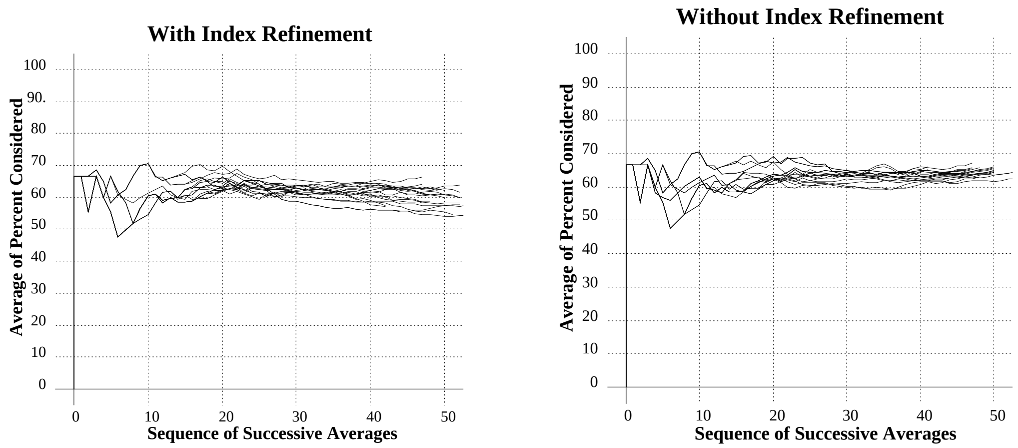

Figure 7.11: Successive averages for the worst anomalous case (Random 5): almost no difference between introspective and non-introspective graphs

Random 5, using the successive averages of percentages considered as we did in Section 7.3.3, we find that the average percentage considered in retrieval did not change significantly with or without introspective learning. The anomalous sequences in the $p < 0.25$ range of probability showed slightly better retrieval performance, and those in the $p < 0.1$ range showed retrieval performance nearly the same as sequences that showed significant differences in performance. The lack of improvement in retrieval efficiency suggests that, for these sequence, the new features learned introspectively did not apply to later situations and so could not provide any benefit in retrieval. Consequently, the features learned through introspective reasoning did not group together useful cases, but rather might have caused an occasional additional failure by restricting retrieval so that a useful case might not be considered at all.

7.6 Conclusions from the empirical tests

These empirical tests of ROBBIE were designed to evaluate the effect of introspective index refinement on ROBBIE's performance of its planning task, to examine the performance of the system across a wide range of circumstances, and to determine how problem order affects the case-based learning of the planner and the combined case and introspective learning of ROBBIE as a whole. To our knowledge, it is the first attempt to quantitatively evaluate the effects of combining an introspective diagnosis and repair system with an underlying reasoning system.

We found, as expected, that index refinement improved the average success rate on a sequence over case learning alone. At the same time that the success rate was increasing, the work done by the Retriever was decreasing as new indexing features permitted a tighter focus on the most appropriate cases for a given situation. This tighter focus, when combined with poor planning performance that caused irrelevant features to be selected for the indexing criteria, could cause over-restriction of the retrieval process so that correct cases were overlooked. In this situation, it is possible that introspective learning might not improve the planning performance of the system.

The fact that over-restriction may arise when the case learning performance is especially poor emphasizes the importance of taking into account the effect of problem order in general on both case-based learning and introspective learning of indexing features. Our examination of this issue led to a definition of a "well-ordered" sequences: one in which

- Early goals should (eventually) span the entire range of possible situations to lay the groundwork for later learning, and
- Each goal in sequence should extend the scope of solvable problem a small

amount, rather than being a large departure.

The sequence we defined to conform to these criteria permitted ROBBIE to perform at a very high level. As the ordering of goals in test sequences became more and more random, ROBBIE's performance degraded, with or without the presence of introspective index refinement. The success rate degraded more slowly when introspective learning was present to assist in categorizing the problems which *were* successfully achieved. We may conclude, therefore, that care should be taken with the initial training of a case-based system that starts with a small case memory. A lack of care may result in extremely poor planning performance, and index refinement that does not provide any benefit over case learning alone.

By comparing the sequence of retrievals made during a test run with and without introspective index refinement, we discovered that new indexing features have the immediate effect of focusing the retrieval process and reducing the percentage of memory considered good matches. We also learned that, after an initial settling-out period, new features tend to be learned after low-percentage retrievals, and some high-percentage retrievals persist throughout the test run. From this we conclude that learned indexing features may focus retrieval without focusing it perfectly: new features learned after low-percentage retrievals probably help to adjust the focus of retrieval to better reflect the important aspects of a situation. We also conclude that when a new problem is sufficiently dissimilar to the cases in memory, a high-percentage retrieval is appropriate. The Retriever is casting a wider net and bringing into consideration more cases, which improves the likelihood of selecting a good case when no case is a spectacular match.

When care is taken by the goal provider of an "infant" planning system such as ROBBIE, we may also conclude that introspective index refinement performs just as we intend it to: the success rate increases, on average, and the extended indexing criteria allow the retrieval process to focus on a few appropriate cases, limiting the work to be done in selecting the best case, and causing better cases to be chosen than the initial criteria permitted.

Chapter 8

Conclusions and Future Directions

> The previous chapters have shown that introspective reasoning can detect reasoning failures and alter the reasoning process to avoid future failures, using a declarative model of the *ideal* reasoning process being examined. We discuss our methods, results, conclusions, and future directions for this research.

This research examines the problem of modeling introspective learning: of representing knowledge about reasoning, and using that knowledge to diagnose failures in an underlying reasoning process and to refine the process to prevent future failures. Our approach uses a model-based reasoner to form expectations about the ideal reasoning behavior of the underlying reasoning system; the model consists of assertions about the ideal state of the reasoner at specific points in the reasoning process. Our introspective framework detects reasoning failures by an ongoing monitoring of the underlying reasoning process, comparing the actual reasoning performance to the ideal expectations of the model. It explains failures by searching from a detected discrepancy for other assertion failures which might have initiated the problem that was detected. The framework is general: introspective models of different underlying reasoning systems may be constructed from the same generic building blocks, and it facilitates re-use of model fragments for introspective reasoning about systems which share commonalities with ROBBIE.

We applied our introspective framework in ROBBIE to an underlying route planner. The planner combines case-based and reactive components: the case-based planner creates high-level descriptions of routes which the reactive component executes in its simulated world. New routes are learned by storing the result of executing a plan. Combining deliberative (i.e., case-based) and reactive planning results in a planner that can view the problem as a whole in deciding on a route, but can also respond quickly to changes during execution. Several other features of the case-based planner are unique: the use of case-based retrieval mechanisms for components of the CBR process, and the use of re-retrieval to permit complex problems to be broken into more easily solved pieces.

The introspective component of ROBBIE monitors the planner's entire reasoning process, and can explain reasoning failures which occur in any component. We focused on one class of repairs to the reasoning process: extending the indexing criteria used by the planner in retrieving old cases. Determining the features to use in indexing criteria is a key problem for case-based systems; ROBBIE learns such features through introspective experience.

Experiments performed to evaluate ROBBIE's performance with and without its introspective learning component show that introspective index refinement causes an improvement in the efficiency of the retrieval process, by permitting consideration of fewer cases. In addition, ROBBIE performs more successfully when using both case and introspective learning than when restricted to just case learning. Potential improvements through the use of introspective learning may not be realized, however, if problem orders pose too many difficult problems for the system.

Our work applying introspective reasoning to planning has raised many issues for future study. Future directions for this research fall into two main classes:

- extensions or alterations to ROBBIE itself to explore incomplete aspects,
- and extension and application of our introspective framework to a broader set of reasoning systems, elaborating the framework as needed.

We must also consider the larger issues this research touches on: the issues involved in systems for complex domains, and broader kinds of introspection about reasoning itself.

In this chapter we discuss the important features of this research, and what we learn from its implementation in ROBBIE. We also discuss the different directions this research may take in extending our understanding of introspective reasoning and its role in the creation of artificial intelligence systems which can reason about their domains and themselves, and can learn to respond flexibly to complex, changing circumstances.

8.1 Domain issues

The domain implemented in ROBBIE's world simulator provides enough complexity to challenge ROBBIE's planner. Given the small initial case base, ROBBIE must often choose cases which are relatively distant matches to its current situation. Because of the complexity of knowledge involved in the description of a given situation, judging which is the best previous case can be problematic. The complexity of ROBBIE's domain is the perfect testbed for introspective reasoning: ROBBIE can learn through experience which features *turn out to be important* for a given set of experiences. Such features may include locations which are pivotal either for their common

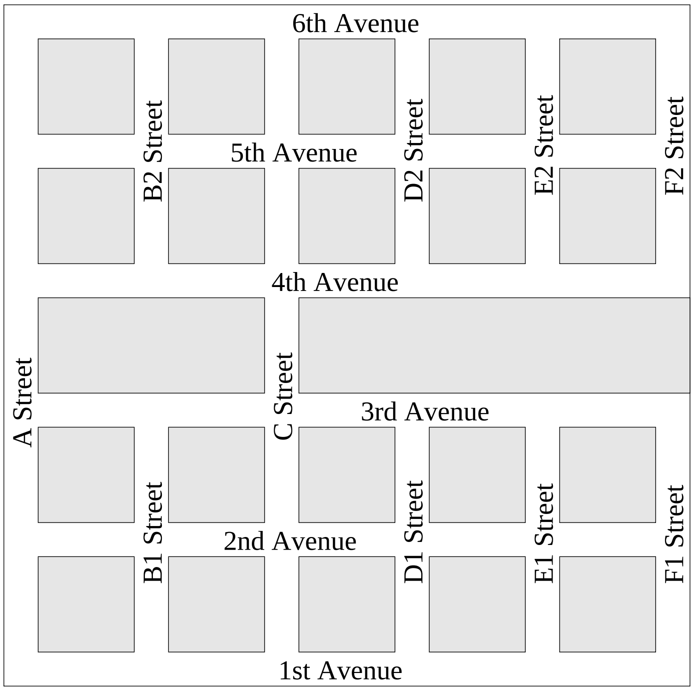

Figure 8.1: The “Bridge” map

appearance or the specialized needs of plans in their vicinity.

We have begun testing ROBBIE in larger and more complex world maps than the three-by-three version upon which we have focused most of our attention. Larger maps permit us to explore a richer set of situations; the map in Figure 8.1 simulates a river through the simulated robot's world. Learning how to select cases which require crossing a bridge may involve alternative introspective learning strategies or qualitatively different features. The feature “crosses river” is more abstract than ROBBIE's current feature types, and is more difficult to detect from route endpoints. The important features for a domain such as the Bridge world differ from those in ROBBIE's ordinary domain; requiring extensions to the feature detection mechanism associated with ROBBIE's introspective reasoner.

Future directions: The domain implemented by the world simulator succeeds in providing a level of complexity requiring introspective learning. Nonetheless, extensions to the domain to include more complexity could be considered. Some extensions are already foreshadowed: the enforcement of blocked streets and other pedestrians as obstacles for the simulated robot.

The representation of time in the world simulator is simplistic, as is the way in which the simulator and ROBBIE interact. A better implementation would permit the simulator and ROBBIE to be autonomous asynchronous processes, in order to represent time in a more realistic fashion.

Exploration of alternative task domains would also extend our understanding of the power and limitations of introspective learning. Such domains should still be constrained by the properties we declared important for ROBBIE's domain: a richness of knowledge about objects in the world, a complexity in the number and form of objects, and dynamic elements which change outside the reasoner's control. Domains that meet these criteria include information-finding tasks such as Web searching agents, complex scheduling problems (i.e., a travel agent, or a package company controller), real-world robot control systems, and many others.

8.2 The planning component

ROBBIE's planner was designed to examine the effects of limited initial knowledge on the success of a case-based planner. In addition, it explores the combination of case-based planning with reactive planning to both generate plans and execute them. It incorporates several unique ideas for case-based planning, which are reflected in the introspective model of its reasoning process. It introduces a process called *re-retrieval*, used to simplify a route goal that is too complex for the planner to handle. Re-retrieval splits the original goal into two more simple goals and constructs solutions for each goal separately. ROBBIE's planner also re-uses its case memory and retrieval process to implement components of the planner itself.

8.2.1 Combining deliberative and reactive planning

ROBBIE's combination of a case-based planner with a reactively executing planner gains the benefits of both approaches. The case-based planner, because it views the problem as a whole, takes into account long-range features affecting the entire route. For instance, if the robot is starting on an east/west street and the goal is to end up on an east/west street three blocks to the north, the case-based planner will select which north/south street to use based on whether it extends all three blocks to the north, rather than selecting the closest one. A wholly reactive planner might choose the nearest opportunity to head north, even if that street does not intersect with the goal street, because the nature of a reactive planner is to focus on the current situation and ignore the long view.

The benefit of a reactive planner is its ability to respond to the unexpected obstacles and dynamically changing elements in its domain. ROBBIE's reactive planner chooses actions quickly, and without looking further than the situation of the moment. If the situation changes in one time step, the actions of the reactive planner will change to respond to it. The reactive planner, then, provides a quick response without looking ahead. The case-based planner does look ahead, but takes longer to generate a plan of action. However, in a real world the cost of execution would dominate planning costs, so the savings from performing good strategic planning are still considerable. A mistake in execution which leads to wasted movements will overshadow the costs of deliberative planning to avoid that mistake (i.e., if the robot had to backtrack).

Because we have combined deliberative and reactive planning, the case-based planner need not generate as detailed a plan as would otherwise be required. An abstract description of the route plan is sufficient, since each step will be interpreted by the reactive planner in terms of the momentary situation. This minimizes the time that must be spent deliberating about the details of the plan, while at the same time ensuring that the reactive planner has enough guidance to avoid problems due to its lack of long-term reasoning. Combining deliberative planning with reactive planning has been studied by various people (Gat, 1992; Nourbakhsh et al., 1995), but ROBBIE is the first instance we know of a *case-based* planner being combined with a reactive planner.

Future directions: While ROBBIE's planning components perform well together, further work could be done to balance the power of the reactive planner against that of the case-based planner. The reactive execution component of ROBBIE currently has the capability to execute any *correct* plan it is given. However, it is limited in how it can respond if the plan it executes turns out to be flawed. The reactive planner should be extended to include the ability to re-plan when the original CBR-generated plan has obviously gone awry. Currently the Executor informs the introspective reasoner of the problem and then gives up trying to execute the plan. A better alternative would be to incorporate reactive strategies for re-applying case-based planning from the current location of the robot, for trying an alternative planning strategy (such as a graph search, or a rule-based approach), or for reactive wandering intended to move generally in the right direction.

At the current time the reactive planner is viewed as a component of the overall planning system. Because the Executor is the component most directly tied to the world simulator, if it were given more power the planner as a whole might be more accurately described as a reactive system with a deliberative planning component available to it. This would be more similar to the approach developed by Gat (1992) in ATLANTIS.

8.2.2 Re-retrieval to adapt goals

ROBBIE starts out with a small number of route plans in its memory. Its main adaptation technique is to map the locations of the new situation onto the locations of the old one and convert the plan steps accordingly. Thus, the new plan always has the same number of steps as the old one. It is possible, therefore, that a situation will arise which will require more plan steps than any case in memory. Any case retrieved for such a situation will be unadaptable because the locations will not map onto each other: one endpoint might require a north/south street while the endpoint of the case might be on an east/west street. In order to permit ROBBIE to recover when retrieval provides an unadaptable case, we developed a re-retrieval mechanism to adapt the original *goal*. The goal is broken into two smaller parts by selecting an intermediate location between the two original locations. Each subproblem is then recursively addressed by case-based planning. The resulting solutions may be concatenated to form a more complex plan. ROBBIE's Re-retriever uses a heuristic method to select the intermediate location. While the heuristic method may fail occasionally, the retrieval of unadaptable cases would otherwise guarantee a failure.

Future directions: Re-retrieval is currently restricted to a single recursive application: if a simplified goal produces an unadaptable case ROBBIE gives up on the problem. This restriction avoids the problem of infinite recursive application of re-retrieval. Development of better heuristics for selecting good intermediate locations, combined with a mechanism to recognize when re-retrieval is not progressing, would permit lifting that restriction. Instances when re-retrieval is not succeeding include when the intermediate location is identical to one of the two original locations; and when one of the simplified goals is as simple as possible and the case retrieved is still unadaptable.

The application of re-retrieval is currently always viewed as a correct part of the reasoning process. However, the un-adaptability of a case may stem from a failure of the indexing criteria to select the appropriate case, not from the complexity of the current problem. ROBBIE does not currently distinguish between these two situations. Further work should be done to determine how to distinguish necessary re-retrievals from re-retrievals caused by reasoning failures, in order for introspective reasoning to correct the retrieval process when it is flawed. The question remains whether every re-retrieval in which cases of similar complexity exist in memory should be considered a failure. It is possible that the cost of adapting a dissimilar but equally complex case is higher than the cost of breaking the goal into two more simple parts and solving each separately. Further investigation of this tradeoff should be done in the future.

8.2.3 Recursively using CBR for components

The case memory for ROBBIE contains cases for the planner itself, and cases for components of the planning process: Indexer, Adaptor, and Executor. Each component uses the Indexer and Retriever recursively to select from memory appropriate structures for performing its task. Re-using case-based reasoning mechanisms to implement parts of the CBR process is an elegant approach: we contain most of the knowledge needed by the system in one structure and access it with one mechanism. It also simplifies the planner by using a single mechanism for similar tasks in different parts of the reasoning process.

In order to implement this recursive use of case-based reasoning, we developed a general retrieval mechanism capable of handling multiple forms of indices, using different similarity measures for different indices, and retrieving different numbers and kinds of cases. This general retrieval mechanism separates the process of retrieval, which remains the same, from the details of retrieving any particular kind of memory object. By developing a generalized retrieval mechanism, we can see which parts of the retrieval process are general, and which depend on the use to which the cases are to be put.

Future directions: Using case-based *retrieval* to implement parts of the case-based reasoning process raises the immediate question of incorporating *adaptation* and *learning* of the components' cases. To what extent adaptation may be applied, and how the Adaptor and Storer must change to accommodate different memory objects is a topic for future elaboration.

Despite the insights we have gained in ROBBIE about generalized retrieval mechanisms, an interesting alternative approach would examine the feasibility of integrating the indexing criteria and similarity assessment mechanisms for different kinds of memory objects. Future work should determine if a similarity measure exists which would apply to all the different cases in memory; if such a measure existed it would bode well for an integrated adaptation mechanism for such cases, since a unified similarity measure must rank the *adaptability* of different kinds of cases.

Extending the kinds of knowledge structures in the case memory would further broaden our understanding of different forms of retrieval, as well as enhancing the elegance of the reasoner by incorporating more of its knowledge into one storage and access model. Other knowledge of the planner includes its knowledge of the streets of its domain, and interpretations of sensory knowledge.

More central to our interest in introspective reasoning, the planner and introspective reasoner could be brought closer together by storing the introspective model as cases in memory as well. This raises the unhappy specter, however, of poor retrieval of introspective cases leading to decreased performance by the introspective reasoner, and hence poorer performance by the system as a whole. Determining the feasibility of case-based modeling of reasoning is an interesting future problem.

8.2.4 Storage of cases

The case memory for ROBBIE contains many different kinds of cases. It includes sophisticated indexing to retrieve cases, but storage in memory is done in a simple and straightforward way. This simple memory storage increases the cost of retrievals, but is not a significant problem for a system with a small case memory. As the case memory grows, however, it becomes more expensive to do retrieval and storage of cases, and more difficult to ensure that the same case does not get stored twice, nor two different cases get assigned the same index. Other research has examined different case storage methods (Kolodner, 1993a), which might be applied with positive results here.

ROBBIE currently stores only successful cases in memory. Altering the memory organization to permit storage of failed cases to provide warning against certain paths in the future could improve ROBBIE's performance when there are more failures than successes, by increasing the cases available in memory for a given situation.

8.3 Introspective learning

Our goals in designing an introspective reasoning component were to create a general framework of wide applicability, to examine introspective reasoning as a failure-driven process, to develop a single-model representation capable of both detecting and explaining failures, and to enable continuous, "on-line" monitoring of the underlying reasoning process. The framework developed for ROBBIE achieves these goals. The building blocks for describing a reasoning process in our framework are generic and designed to correspond to features many reasoning systems have: components, abstract versus specific descriptions, knowledge structures, and so forth. Introspective learning occurs when the assertions of the model about the ideal reasoning process fail to be true of the actual processing. The model contains constructs which permit quick access to assertions to monitor for assertion failures, and constructs which permit the model's assertions to be searched in order to explain detected failures by determining other related assertion failures. Quick access to relevant pieces of the model for monitoring and the ability to suspend assertions which cannot be evaluated due to information constraints permit monitoring to occur in parallel with the actual reasoning process. In this section we examine more closely our main goals for the introspective framework.

8.3.1 Generality of framework

Using a declarative model to describe the ideal reasoning process is the first step in ensuring that the framework developed is one generally applicable to different underlying systems. The assertions in the model are described using a vocabulary of predicates carefully designed to be independent of ROBBIE's own requirements. The assertion vocabulary allows arguments to predicates to be implementation-specific; in that way specific knowledge structures may be referred to by an assertion without a losing the generality with which the assertion is defined. The structure of the model uses generic building blocks like "abstraction," "next," and "component" to organize the assertions. In addition, the model uses highly modular clusters to separate assertions which are general and perhaps re-usable from those which are specific to a given system.

Future directions: As an extension to the model-based theory of introspective reasoning, and in order to test the generality of ROBBIE's introspective framework, we must use it to construct models for introspective reasoning about other underlying systems. A first step would be to add an introspective component to another case-based planner, such as micro-CHEF. Other non-case-based systems should follow. An interesting application would be to construct a model for the introspective reasoning process itself. Using introspective reasoning on the introspective reasoner itself would be a step towards a reflective reasoning system. We would expect to refine the framework as the different requirements of other underlying systems illuminate limitations or new problems that were previously unidentified.

8.3.2 Monitoring and explaining failures

The model-based approach to introspective reasoning which we developed focuses on failure-driven learning. This limits the cost of introspective reasoning by restricting its main effort to those times when it is known that an improvement to the underlying system is possible. Our introspective reasoner expends a small effort throughout the processing of the underlying system in order to look for expectation failures. The structure of the model of reasoning ensures that the monitoring task requires as little effort as possible by making it easy to find the currently relevant assertions. Only when a failure is known to have occurred does the introspective reasoner expend a larger effort by searching for other assertion failures that relate to the originally detected one to determine how the failure came about.

The model supports both monitoring and explanation of failures by maintaining two ways of organizing the same set of assertions. Assertions are clustered by their specificity and the component to which they refer, permitting assertions relevant to the current point of the reasoning process to be found quickly. Assertions are also linked together according to their causal relationships to each other, regardless of the cluster to which they belong. The search for other assertion failures to explain a detected one follows these causal links to find the most likely sources for a given failure. Other introspective reasoning systems tend to stress either detection of failures or their explanation; our intent was to develop a single model and approach to introspective reasoning capable of addressing both tasks.

Future directions: We have performed experiments which showed that the introspective reasoning process improved ROBBIE's performance over case-based learning alone. However, further examination of ROBBIE's learning could be done to determine if its introspective learning is applied when we expect it, and if it always learns the right thing. Further experiments could examine in more detail the costs of introspective reasoning as well as the circumstances in which it produces an improvement.

The planner keeps a reasoning trace which describes the reasoning that has occurred up to the current point. The introspective reasoner uses that reasoning trace in a limited way, so far, to determine the values of knowledge structures to which it needs access. To extend the power of our approach to introspective reasoning, the use of the reasoning trace should be expanded so that, if necessary, the introspective reasoner could reconsider every retrieval made, every adaptation strategy applied, every detail of the underlying reasoning. The method used to control the search for failed assertions when explaining a detected failure requires further study to fine-tune its focus on important causally-related assertions. In addition, the kinds of links in the model cover most kinds of causal relationships between assertions, but further refinement may be desirable.

Developing a working framework for introspective diagnosis and explanation of reasoning failures has been the focus of this research. The most important extension to that introspective framework is the addition of a general approach to repairing reasoning failures. For a general approach, we must be able to describe, in generic terms, different repair strategies, and a generic description of repairs to many different components of the underlying system. Because of the importance of indexing feature selection for case-based reasoning systems, ROBBIE got a great deal of mileage out of a single repair strategy. However, failures occur which cannot be traced back to a retrieval problem. These failures are currently detected and explained; the introspective reasoner should be able to repair them as well. A broad, generic repair

Class	Description
`add-case`	Construct a new memory object which addresses a gap in the reasoning process
`change-value`	Alter a numeric value to increase or decrease some behavior
`alter-structure`	Change an existing knowledge structure to correct a flaw in it
`delete-case`	Remove a bad case from memory to prevent further use

Table 11: Current repair classes for the model

module is a necessity if the framework is to be applied to differing underlying systems.

Preliminary work has been done in determining a partial catalog of repair types, which have been included in the model as unimplemented suggestions (Table 11 lists the current catalog). A mechanism for selecting a repair strategy, given a description of the failure and a suggested repair class, is currently implemented in ROBBIE, with only one strategy included (`add-case`, used for index refinement). The possible repair strategies must be classified in terms of the kinds of changes they make to the underlying system, in order to develop a generic vocabulary for describing repairs and generic mechanisms for interpreting those descriptions.

8.4 Index refinement

The application of introspective reasoning to refine the indices of the case-based planner addresses one of the key issues for designers of case-based systems. The success of a case-based reasoning system depends on its ability to retrieve from memory the best applicable case; that case should be the one most easily adapted to fit the current situation. Thus the decision about what features to use in indexing cases in memory should be tied to predictors of adaptability. Our approach, allowing the features to be determined from experience, links the indexing criteria for plans very closely to success in adapting and applying the case. New features will be added when existing features fail to retrieve the most easily adaptable case (which is defined for ROBBIE as the case with the most similar plan steps). The burden on the system designer of selecting features for indexing is lessened, and the choice of indexing criteria is guided by the actual application of cases to real problems for the system.

Our experiments have shown that the new indices learned introspectively capture interesting and useful features of the situations ROBBIE faces. The addition of new indices driven by retrieval failures makes ROBBIE more successful in creating plans to reach goals. New indices also permit the retrieval process to focus on smaller sets of cases by grouping cases which share the same new feature together.

Future directions: The first direction of future research into index refinement must be further study of the tendency of introspective index refinement to over-restrict the retrieval process. When the deck is stacked against the learner, i.e. when the planner is repeatedly unsuccessful in achieving goals, the case memory remains small. The smaller the case memory is, the less likely it is that the solution of a case will match closely with a new problem's solution. The closest case in memory may not share a particularly useful feature with the new solution, causing the introspective reasoner to add a useless feature instead. Each useless feature added to the indexing criteria will tend to cause the retrieval process to focus on smaller sets of cases, just as a useful feature would. The result may be over-restriction of the retrieval process as potentially applicable cases, which happen not to share a useless feature included in the indexing criteria, are ignored in favor of less applicable cases, which do have the useless feature. This phenomenon appeared in the empirical tests we performed on

ROBBIE — another benefit of performing empirical tests — and must be studied further to determine how to alleviate the phenomena.

The method used in ROBBIE to determine what relevant features of a situation are missing from the indexing criteria is based on a set of heuristic feature types. For each feature type a simple test exists which determines whether the feature is present in a given case. This method of determining features for the indexing criteria is not generic: it is limited to the feature types we chose to include. We intend to examine more principled methods for determining missing features. One such method would be to apply explanation-based generalization to explain what features are shared between two cases whose plan steps are similar but whose indices fail to reflect that similarity. An explanation of the relevance of the potential features of a situation could determine which features are especially important for a given case.

8.5 Scaling up

ROBBIE's domain incorporates the important features for a domain to stimulate introspective reasoning: it includes knowledge-rich and dynamic objects with which ROBBIE must contend. It is, however, relatively simple compared to many interesting real-world domains (as we described above). In addition, the domain knowledge of ROBBIE remains relatively small; ROBBIE's knowledge generally stays below 200 cases in memory. A natural question is how the approach tested in ROBBIE would "scale up" to a larger problem involving a large case library and a richer task and world. There are two related aspects involved in scaling ROBBIE's approach: changes needed to scale up the planning component, and changes needed to scale up the introspective learning component. In this section we describe the effects of increasing the problem complexity on each component of our approach and in the next section we illustrate our ideas with an example of a highly complex domain to which case-based planning with introspective learning could be applied.

In terms of handling more knowledge and more complex situations, the key portion of ROBBIE's overall processes is the case retrieval process. Other operations both in ROBBIE's domain and introspective tasks depend upon case retrieval or are otherwise bounded in some way, even when the task complexity and knowledge involved increase in difficulty. The cost of introspective learning depends on the complexity of the reasoner involved, not its domain, task, or knowledge. A relatively small model can capture sufficient expectations about the reasoning system to provide a benefit to its performance, as ROBBIE has shown. If introspective reasoning is bounded in its size and cost as the problem complexity increases, then the question remaining is how the planning process performs. Case retrieval is central to the planning process as a whole, and it is central to the processing of other components which use case retrieval to access the knowledge structures they require.

8.5.1 Scaling up the planning process

Case retrieval is central to many *components* of the planning process, as well as the whole task of plan generation. Therefore it is no surprise that the costs associated with retrieving cases from memory are the most important factors in extending ROBBIE's planning approach to more complex domains. To examine all the costs of scaling up ROBBIE's planner, we examine each component of the planning process in turn.

Index creation

As the situations a system like ROBBIE faces become more complex, so do the indices which describe the relevant features of each situation. The Indexer component of ROBBIE depends on quick retrieval of indexing rules from memory to build a complete index. The costs of building a basic index form and applying the Indexer recursively would remain relatively constant as complexity increases. If rules can be quickly retrieved from memory, the cost of applying them is proportional to the number of rules applied to a single index, which should never become very large.

Case retrieval

Retrieving the right case efficiently depends on two issues, one of which ROBBIE has addressed. The first issue is determining the right features to use to decide which cases in memory are similar: ROBBIE uses introspective learning to help determine these features. The second issue is how actual cases are accessed in memory. ROBBIE uses a simple case storage method: retrieving or storing cases in memory requires a linear examination of every case in memory. Such a simple approach would become too expensive if the number of cases increased dramatically. More sophisticated case storage methods have been examined in other research and are applicable to ROBBIE's case library: discrimination networks (Hinrichs, 1992; Koton, 1989; Simpson, 1985), implementation on parallel machines (Evett, 1994; Stanfill & Waltz, 1988), and multi-stage retrieval methods (Kolodner, 1988; Simoudis, 1992) (see Kolodner (1993b) for a full discussion of case storage methods). These methods are made effective by the use of good features in indexing, which ROBBIE's index refinement facilitates.

Organizing ROBBIE's cases with discrimination trees: With a serial approach to case retrieval, collecting cases that share certain features in common can improve retrieval efficiency. The case memory can be partitioned into groups, selecting a case becomes a process of selecting more and more restricted partitions of memory. Retrieval can focus on only the subset of cases similar to the goal index without examining individual cases and their full indices. A variety of approaches to discrimination networks implement this kind of solution. To take ROBBIE's own case memory as an example, we could group cases first by their type (`plan`, `indexer`, `adapt`, or `planlet`). We could further group plan cases according to starting streets, or special features, or other features that plans might share in common. We could then eliminate all cases of different types and different starting streets without examining the indices of individual cases at all.

A discrimination network of cases in memory is constructed as cases are added to memory. A new case will cause changes to the discrimination structure in order for the new case to be distinguished from existing ones. Using a discrimination network requires that each goal index contain the information needed to make a discrimination at any particular point in the network. If information is missing, discrimination may be difficult (e.g., if ROBBIE did not know the starting street for a goal). The discrimination network must be as full as possible as well, for optimal performance. In the best case, a discrimination network approach would permit case retrievals at $\mathcal{O}(log_b n)$ where n is the number of cases and b is the branching factor at each discrimination point, but in the worst case retrievals would be $\mathcal{O}(n)$. The worst performance would occur when each question removed only one case from consideration.

Using parallel retrieval methods: Because of the overhead of maintaining a discrimination network and its potential worst-case cost, research into parallel implementations of case retrieval has provided an alternative approach. Evett (1994) implement in PARKA a knowledge retrieval method on a Connection machine that guarantees performance at $\mathcal{O}(d+p)$ where d is the hierarchical depth of knowledge (from an individual case to abstractions of it), and p is the number of features in the desired index. This retrieval performance is independent of the number of cases in memory. Other approaches use multi-stage retrieval processes in conjunction with a potentially parallel implementation. Cases are evaluated using a simple estimate of similarity, a process which could be performed in parallel, and deeper evaluation is performed only on the most similar cases (Kolodner, 1988; Simoudis, 1992).

Large case memories are not uncommon in industry applications of CBR. Even without expensive parallel hardware implementations, accessing cases in a large memory can be performed efficiently and quickly by combining case-based reasoning with existing relational database technology. Kitano et al. (1992) access cases using by posing queries to a relational database containing, at last count, over 20,000 cases. Cases describe solutions for problems in software quality control, and are used interactively by users attempting to solve problems.

Starting from indices that contain the right set of features is the key to high-quality, efficient retrieval. Approaches that dynamically restructure the case memory would benefit immediately from ROBBIE's ability to learn the relevant features for indexing cases. Restructuring of cases in memory often occurs when a new case is learned, and involves only that case and a few neighbors. Introspective learning of new indexing features would trigger a reorganization of the entire memory and would improve the parameters by which cases are chosen.

Case adaptation

As for the Indexer above, the cost of retrieving adaptation strategies dominates the cost of performing adaptation. Other costs associated with adaptation are controlled. ROBBIE's adaptation component is general enough to apply to many different problems. The Adaptor is highly dependent on case retrieval to select the right adaptation strategies for a given situation. Besides retrieval costs already discussed, the only significant cost in ROBBIE's adaptation process is selecting what portion of the case to adapt. Determining what changes to make during adaptation remains an open question, although learning when to apply a given adaptation strategy, or even learning new strategies, could help to limit the adaptation effort when the case being adapted is highly complex (Leake, 1995).

In general, as the case library for a case-based reasoner grows in size, the need for extensive adaptation decreases: a case selected will be more similar than when few cases are available. In addition, ROBBIE's introspective learning helps to focus the retrieval process, reducing the effort expended in adaptation by selecting the best possible case. For many practical applications of CBR in complex domains, adaptation is not a requirement. Kitano et al. (1992) implement in SQUAD a system to assist a human user in diagnosing and correcting software problems, without automatic adaptation. CLAVIER proposes layouts for baking high-technology parts in an autoclave, but adaptation of old solutions is left to a human user (Hennessey & Hinkle, 1991).

Case applications

Given that a case-based plan is to be executed interactively, there are two aspects of the reactive planning process that pose obstacles to scaling up. The first is the need to quickly select a new action given a new context. Because ROBBIE uses CBR for this problem, this reduces once again to a case retrieval problem, which ROBBIE already begins to address. The second aspect is performing a fast analysis of the current situation to determine whether the current goal has been reached or the context has changed. Analysis of its domain was straightforward for ROBBIE because its sensory input was pre-processed; a real robot would require a component to perform this pre-processing. Currently existing robot technology can make good estimates of distances to nearby objects and can locate distinctive objects in visual range (e.g., (Firby et al., 1995)).

Case learning

In order to create cases for storage in memory, a system such as ROBBIE that includes both case-based and reactive reasoning must be able to reconstruct a case from actions of its reactive component. This process will become more complex as the domain complexity grows. ROBBIE has a set of rules which reverse the mapping made from high level steps to robot actions; these rules would be more numerous and perhaps more involved (looking for retraction of some actions, for example). However, as case learning occurs after the solution to a given problem, reconstruction of the final solution does not face the same time pressure as reactive application of a case, and could be performed during otherwise idle time.

8.5.2 Scaling up introspective learning

The question of domain complexity has little effect on the requirements of the introspective learning component. The introspective reasoner does not reflect the complexity of the domain, but the complexity of its underlying reasoner. While a domain may grow more complex, the reasoning system that performs in that domain may not significantly increase in complexity.

The cost of monitoring for reasoning failures is relatively constant across the underlying reasoning process. Because the organization of ROBBIE's model helps to focus attention on just the relevant portions of the model as it is examined, ROBBIE's introspective reasoner evaluates less than ten assertions at any given point in the reasoning process unless a failure is detected; a system for a more complex domain should maintain a similar level of detail. The time spent on introspective reasoning is small compared to the time spent planning, and both are small compared to the time spent actually executing a plan. The overall time required by ROBBIE is much less than a physical robot would require moving in a physical world. Further refinement of the criteria for assertion relevance could improve the monitoring process even more.

Explanation of a detected failure takes much more time than monitoring to detect failures. The explanation process will, at worst, evaluate every assertion in the model, although in practice many assertions are not relevant to a given detected failure. So long as the model remains similar in size to ROBBIE's current model, however, the time spent in explanation is a reasonable price to pay for the benefits of introspective learning. If time pressures warranted it, the search process itself could be refined further to focus its search more carefully, limits could be placed on it to prevent it from taking too much time, or a parallel implementation could evaluate many model assertions at once, as model dependencies allow.

8.6 Web searching as a real-world application

In the previous section we described the issues involved in scaling up ROBBIE's approach to any more complex domain. In this section we describe in detail how ROBBIE could be applied to one particular domain mentioned earlier: a system for assisting a user in searching the World Wide Web for information.[1] While in principle the introspective reasoning framework, the heart of our research, could be applied to an underlying system that takes a very different approach than case-based reasoning, for the purposes of this example we assume that the Web searcher does use the general approach of ROBBIE: case-based planning combined with plan execution. In this case, the plans would be methods for finding the requested information, and plan execution, for simplicity, could be left to a human user.

A ROBBIE-like Web-searcher — maybe Charlotte would be a good name — would follow the basic division between task and introspective processing, with an introspective reasoner modeling an idealized reasoning process and examining its actual reasoning process for deviations from its model. The task itself would be approached in much the same way: Charlotte would be given a description of the kind of information to be found, for instance "Information about podiatry schools in Minnesota." It would generate a plan for going about finding that information, then use that plan to suggest possible avenues of exploration to its user. It would store the user's resulting search, along with an evaluation of its quality provided by the user. With a more complex problem domain like web-searching, determining whether a proposed plan was successful is more difficult than in ROBBIE. The user's input as to a given plan's utility might be needed for Charlotte to fully rate a search plan's success.

[1] We have chosen this domain as an example, but we make no claims to a deep understanding of existing Web searching technology.

In the following sections we break the task down into ROBBIE-like components to describe how each part of the problem would be addressed both at the task level and the introspective level.

8.6.1 Index creation

Whereas in ROBBIE index creation for plan cases is initially simple and depends on introspective learning for any specialization, index creation for Charlotte would be much more important and would require more elaboration of the original problem description from the start. For instance, if the problem were as described above: `Podiatry schools in Minnesota`, Charlotte might want to elaborate the index to include more general information like `medicine`, `specialties`, `location`, `in USA`, and so forth. This sort of elaboration is necessary if Charlotte is to assist with searches for many different kinds of information. No matter how many cases Charlotte has in memory, retrieving search plans based only on the original keywords is not feasible. While Charlotte may have no previous plans regarding information about podiatry, it may have plans for location-specific information, or for information about medicine in general. In addition, good search techniques should dictate the use of *concepts* instead of keywords for searching. Elaborating beyond the specific terms used in a particular query should improve both case selection and later search (Kolodner, 1984; Lehnert, 1978). This sort of elaboration of the original situation seems well-suited for the kind of case-retrieval approach ROBBIE uses.

From an introspective standpoint, Charlotte's model of its index creation process would be very similar to ROBBIE's. Charlotte might include index-elaborating rules of more complexity than ROBBIE's. For instance, a rule might describe searching an abstraction hierarchy for a more general concept to include in the index. These complexities would need to be reflected in Charlotte's introspective model, but constitute a relatively small increase in the model's complexity.

8.6.2 Case retrieval

Search plans in memory for a system like Charlotte might describe the paths taken in previous attempts to find information, relating why each path was used, as well as what the path was. For instance, if Charlotte had previously looked for information about elementary school education in Burlington, Vermont, a search plan might describe starting at a description of sites in the US because a location was specified, specializing to sites in Vermont because that is the location desired, then looking for descriptions of Burlington or school districts.

Whereas in ROBBIE a single route plan could be used to successfully achieve a goal without human intervention, for Charlotte a better approach might be to produce a set of different approaches generated, perhaps, by stressing different aspects of the original query to be provided to the human user. For example, some plans might stress the geographical aspects of our sample query, others might look for education-oriented search points, and others focus on medical searches. Many case-based system designed for assisting human users similarly suggest multiple possibilities and let the user decide which to use. The CLAVIER system assists a user in positioning pieces to be baked in an autoclave; it provides the user with a set of the most similar, unadapted cases in its memory (Hennessey & Hinkle, 1991). CASCADE contains cases describing VMS operating system failures and provides the closest matching alternatives to the user in the face of a new problem (Simoudis, 1992).

Introspective evaluation of case retrieval in Charlotte must evaluate expectations about the applicability of different search approaches, and whether all applicable search plans have been chosen. These expectations are not qualitatively different than the expectations ROBBIE itself has about its retrieval process. Because multiple cases are to be retrieved, expectations should evaluate the breadth of possible search types typified by the set of retrieved cases.

8.6.3 Adaptation

To some extent adaptation of cases may be left to a user in a system designed to support the user's own actions: Charlotte could simply present the search plans it found and let the user decide how to apply them. However, adaptation of search plans could also be performed in a manner similar to ROBBIE's, by substituting for the appropriate differences. To search for medical schools in Minnesota, starting from the search plan for elementary schools in Vermont, Charlotte could substitute the features of podiatry schools for those of elementary schools, and substitute the location `Minnesota` for the location `Burlington, Vermont`. ROBBIE's approach to adaptation is suitable for many different kinds of cases, but Charlotte might also require an information-gathering approach to adaptation more similar to that described in Leake (1995). In such an approach, adaptation strategies include directions on how to search for the appropriate information to substitute, and may be learned through experience as well.

Introspective assertions about adaptation might expect that only valid Web sites are referred to, that there will be links from the starting page to the relevant information, that the needed information for adaptation will be found. These expectations are similar to the expectations ROBBIE has about its adaptation process.

8.6.4 Learning new search plans

The selection of particular search plans to apply and their execution may be left in major part to the human user, with Charlotte keeping track of what the plans said to do and what the user has actually done. From an analysis of the outcome, which could be augmented by user input about the utility of Charlotte's search plans, Charlotte could determine whether its proposed plans were appropriate and useful, and could learn new search plans based on what the user actually encountered. Because the Web is a changing environment, Charlotte should be particularly aware of expectation failures such as new sites or non-existent old sites. Such failures might indicate reasoning failures, or might not, but should be noticed and examined by the introspective reasoner regardless.

As for ROBBIE, once a path is found to the desired information, Charlotte would have more to work with in terms of evaluating its reasoning performance. It should expect only minor alterations to its suggested plans: major alterations or the failure of a suggested plan to find any relevant information should indicate a reasoning failure. Introspective reasoning should have expectations about how the suggested plans and the actual plans should relate. Learning new indexing features or criteria for selecting different search paths could be driven by discrepancies between the actual successful path and the original one. For example, if Charlotte proposed finding information on podiatry schools by looking for specialized institutions, by analogy to dentistry or optometry schools, and that plan found no matches because podiatry is a specialty taught within medical schools, then Charlotte could alter its reasoning to stress the `specialty` feature more strongly. Relating this change to the fact that a *school* is desired might be important; finding information about *podiatrists* might depend on looking for specialized institutions (i.e., podiatry clinics).

Charlotte is a proposed system engaged in a much more complex and dynamic domain than ROBBIE. By examining how ROBBIE's approach could map onto such a complex problem, we can see what parts of ROBBIE's approach are most general, and how we might go about scaling up both the planning and introspective components of ROBBIE to problems of practical interest. Charlotte can use many of the same mechanisms as ROBBIE, although scaling up to Charlotte's needed case memory and knowledge requirements would involve some of the changes discussed in Section 8.5. Converting the introspective component requires the least alteration to handle scaling up, as the *reasoning* of Charlotte is not much more complex than the *reasoning* of ROBBIE. This example demonstrates the wide applicability of our approach to planning and introspective learning: the potential of a case-based planner to successfully encounter complex, knowledge-rich domains, and of introspective reasoning to assist such a system in operating in such a domain.

8.7 Summary

This research has successfully integrated introspective reasoning with an underlying case-based planning system and has shown through empirical tests that the combination outperforms the planner alone. We have developed a general framework for constructing reasoning models for the task of diagnosis and repair of reasoning processes. We have examined the implications of limited initial knowledge on a case-based learning and introspective learning system, the effects of different problem orderings, and the effects of continued limited knowledge if learning does not progress well.

We have also contributed to a better understanding of the more general problem of modeling reasoning processes. We found it necessary to deviate from previous models which were primarily abstract in form to incorporate both abstract knowledge about the reasoning and knowledge that described how the reasoning process actually worked. Abstract knowledge is required for the system to distinguish the big picture from the messy details, and to understand the structure of the reasoning it is describing. Specific knowledge is needed to relate the observed reasoning behavior to the more abstract portions of the model, and to provide details for detecting and repairing reasoning failures.

The ultimate goal we described in Chapter 1 was the development of a learning system that applied the same set of learning mechanisms to opportunities to learn about its domain or itself. The immediate goal of this research was one step towards that long-range aim: to explore the possibilities of using a model-based approach to introspective learning in combination with an underlying domain-knowledge learning system. ROBBIE's successful integration of introspective learning with its case-based learning system, forms a basis from which to generalize the application of introspective learning in artificial intelligence systems.

Appendix A

Sample Run of ROBBIE

This appendix contains the output of a sample run of ROBBIE. Selected portions of this example is used to illustrate portions of ROBBIE's planner; this is a more complete and contiguous set of output (tedious details have been removed). Routine output from the introspective reasoner has been suppressed, introspective output only appear if a failure is detected.

```
--------------------------------------------------
Starting up World and Robot
Time of day randomly set to 11:00:01

Robot is facing east at the position 20 feet along maple street

------------------------------------------
Time = 11:00:01
```

ROBBIE receives a goal and creates a plan index for it:

```
Rob(CBR)>> New destination is:
The east side of elm, the 200 block, 90 feet along
Rob(Ind)>> Creating plan index
Rob(Ind)>> Creating index rule index
Rob(Ind)>> Index is:
(indexer plan 40 23 23 230)
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
```

```
Rob(Ret)>> Retrieval level at 2
Rob(Ret)>> Retrieval level at 1
Rob(Ret)>> Retrieval level at 0
Rob(Ret)>> Cases under consideration are:
  ()

Rob(Ind)>> Index is:
(plan (sidewalk (street maple) (side north) (block 100) (along 20) (across 3))
      (sidewalk (street elm) (side east) (block 200) (along 90)))
```

ROBBIE retrieves a case matching the goal:

```
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case old1 matched with difference value of 40
Rob(Ret)>> Case old2 matched with difference value of 25
Rob(Ret)>> Case old3 matched with difference value of 25
Rob(Ret)>> Cases under consideration are:
  (old1 old2 old3)
Rob(Ret)>> Selecting case: old2
```

ROBBIE applies adaptation strategies to alter the old case
(most individual adaptations will be omitted.)

```
Rob(Ada)>> Considering adaptation of plan old2
Rob(Ada)>> Plan needs adaptation to repair differences
Rob(Ada)>> Mapping for cases found, beginning adaptation loop

Rob(Ada)>> Selecting next point of adaptation:
((1 ((starting-at none)
     (turn none)
     (move-on none)
     (move-to none)
     (turn none)
     (move-on none)
     (move-to none)
     (ending-at none)))
 (0 (starting-at none)))

Rob(Ind)>> Creating adaptor strategy index
Rob(Ind)>> Index is:
(adapt info new-dirs (index 0) major)
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case adapt45 matched with difference value of 4
Rob(Ret)>> Case adapt22 matched with difference value of 1
Rob(Ret)>> Cases under consideration are:
  (adapt45 adapt22)
Rob(Ret)>> Selecting case: adapt22
Rob(Ada)>> Applying strategy:
(map (look-up old-dirs ?ind))
```

Adaptation continues similarly, only selected adaptations will be shown

```
...
```

An example of the selection of an adaptation point which cannot be adapted:

```
Rob(Ada)>> Selecting next point of adaptation:
(...)

Rob(Ind)>> Creating adaptor strategy index
Rob(Ind)>> Index is:
(adapt start copy (on none) turn)
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case adapt5 matched with difference value of 0
Rob(Ret)>> Case adapt45 matched with difference value of 4
Rob(Ret)>> Cases under consideration are:
  (adapt5 adapt45)
Rob(Ret)>> Selecting case: adapt5
Rob(Ada)>> Applying strategy:
(if (blank inter-val) (no-op) (value-of inter-val))
Rob(Ada)>> NO-OPING

...
```

An example of the adaptation of a location for the new plan by copying the location following it:

```
Rob(Ada)>> Selecting next point of adaptation:
(...)

Rob(Ind)>> Creating adaptor strategy index
Rob(Ind)>> Index is:
(adapt info new-locs end 6 blank blank fixed)
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case adapt47 matched with difference value of 7
Rob(Ret)>> Case adapt23b matched with difference value of 2
Rob(Ret)>> Cases under consideration are:
  (adapt47 adapt23b)
Rob(Ret)>> Selecting case: adapt23b
Rob(Ada)>> Applying strategy:
(value-of (next-value new-locs ?ind))

...
```

An example of the adaptation of a street for the new plan by taking the intersection of the sets of streets at the current index and for the location following it:

```
Rob(Ada)>> Selecting next point of adaptation:
```

```
(...)

Rob(CBR)>> Creating adaptor strategy index
Rob(CBR)>> Index is:
(adapt info new-streets move-to 3 fixed filled fixed)
Rob(CBR)>> Case adapt47 matched with 7
Rob(CBR)>> Case adapt24 matched with 1
Rob(CBR)>> Cases under consideration are:
  (adapt47 adapt24)
Rob(CBR)>> Selecting case: adapt24
(find-inters
  (set-intersect
    (look-up new-streets ?ind)
    (next-value new-streets ?ind)))
Rob(CBR)>> set-inters: (move-to (elm) (oak) (cherry) (date)) and
                       (turn (elm))

...
```

An example of adapting a direction of movement for the new plan itself, mapped directly from the old plan's value (the last adaptation strategy applied):

```
Rob(Ada)>> Selecting next point of adaptation:
(...)

Rob(Ind)>> Creating adaptor strategy index
Rob(Ind)>> Index is:
(adapt inter simple (dir none) major)
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case adapt8 matched with difference value of 0
Rob(Ret)>> Case adapt45 matched with difference value of 4
Rob(Ret)>> Cases under consideration are:
  (adapt8 adapt45)
Rob(Ret)>> Selecting case: adapt8
Rob(Ada)>> Applying strategy:
(map old-plan-value)

Rob(CBR)>> Adapted case:
(((across 3 unspec))
 ((street elm birch)
  (side east north)
  (block 200 100)
  (along 90 1))
 ())
((starting-at
   (dir any)
   (on (sidewalk (street maple) (side north) (block 100) (along 20) (across 3))))
 (turn (dir west)
       (on (sidewalk (street maple) (side north) (block 100)
                     (along 20) (across 3))))
```

```
 (move (dir west)
       (on (sidewalk (street maple) (side north)))
       (to (sidewalk (street elm) (side east))))
 (turn (dir north) (on (sidewalk (street elm) (side east))))
 (move (dir north)
       (on (sidewalk (street elm) (side east)))
       (to (sidewalk (street elm) (side east) (block 200) (along 90))))
 (ending-at
   (dir any)
   (on (sidewalk (street elm) (side east) (block 200) (along 90)))))
```

ROBBIE, having created a plan, starts reactive execution:

```
Rob(Exe)>> No current step, getting from plan
Rob(Ind)>> Creating planlet index
Rob(Ind)>> Index is:
(planlet
  (starting-at
    (dir any)
    (on (sidewalk (street maple) (side north) (block 100) (along 20) (across 3))))
  ((facing east)
   (moving #f)
   (mov-speed none)
   (loc (sidewalk (street maple) (side north) (block 100) (along 20) (across 3))))
  ())
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case planlet0a matched with difference value of 10
Rob(Ret)>> Case planlet0c matched with difference value of 10
Rob(Ret)>> Cases under consideration are:
  (planlet0a planlet0c)
Rob(Exe)>> Choosing between planlets:
(planlet0c planlet0a)
Rob(Exe)>> Selecting planlet: planlet0a
Rob(Exe)>> Executing starting-at

Rob(Exe)>> Come to a stop

Robot is currently stationary...
Robot is facing east at the position 20 feet along maple street

-------------------------------------------
Time = 11:00:02
```

Having verified its initial position, the Executor starts the next step, a `turn` *to the west*

```
Rob(Exe)>> Step successful!!
Rob(Ind)>> Creating planlet index
Rob(Ind)>> Index is:
(planlet
```

```
  (turn (dir west)
        (on (sidewalk (street maple) (side north) (block 100)
                      (along 20) (across 3))))
  ((facing east)
   (moving #f)
   (mov-speed none)
   (loc (sidewalk (street maple) (side north) (block 100) (along 20) (across 3))))
  ())
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case planlet1 matched with difference value of 10
Rob(Ret)>> Case planlet2 matched with difference value of 10
Rob(Ret)>> Cases under consideration are:
  (planlet1 planlet2)
Rob(Exe)>> Choosing between planlets:
(planlet1 planlet2)
Rob(Exe)>> Selecting planlet: planlet2
Rob(Exe)>> Executing a turn to the west on maple

Rob(Exe)>> Come to a stop
Rob(Exe)>> Turning west

Robot is currently stationary...
Robot is facing west at the position 20 feet along maple street
```

The traffic signal at Elm and Apple is temporarily broken:

```
Creating blockage
(breakage
  (signal
    (traflight (street1 elm) (street2 apple) (green e/w) (time 17) (warn #f)))
  (start 3)
  (end 33))

------------------------------------------
Time = 11:00:03

Rob(Exe)>> Step successful!!
Rob(Ind)>> Creating planlet index
Rob(Ind)>> Index is:
(planlet
  (move (dir west)
        (on (sidewalk (street maple) (side north)))
        (to (sidewalk (street elm) (side east))))
  ((facing west)
   (moving #f)
   (mov-speed none)
   (loc (sidewalk (street maple) (side north) (block 100) (along 20) (across 3))))
  ())
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case planlet3 matched with difference value of 10
```

```
Rob(Ret)>> Case planlet3a matched with difference value of 10
Rob(Ret)>> Case planlet3b matched with difference value of 10
Rob(Ret)>> Case planlet3c matched with difference value of 10
Rob(Ret)>> Case planlet3d matched with difference value of 10
Rob(Ret)>> Cases under consideration are:
  (planlet3 planlet3a planlet3b planlet3c planlet3d)
Rob(Exe)>> Choosing between planlets:
(planlet3 planlet3a planlet3b planlet3c planlet3d)
Rob(Exe)>> Selecting planlet: planlet3d
Rob(Exe)>> Executing a move west on maple

Rob(Exe)>> Move along

Robot is currently in motion...
Robot is moving west to the position 17 feet along maple street

------------------------------------------
Time = 11:00:04
```

The robot will continue to move west until it reaches the corner of Elm and Maple

```
Rob(Exe)>> Continue current step
Rob(Exe)>> Move along

Robot is currently in motion...
Robot is moving west to the position 14 feet along maple street

------------------------------------------
Time = 11:00:05

Rob(Exe)>> Continue current step
Rob(Exe)>> Move along

Robot is currently in motion...
Robot is moving west to the position 11 feet along maple street

------------------------------------------
Time = 11:00:06

Rob(Exe)>> Continue current step
Rob(Exe)>> Move along

Robot is currently in motion...
Robot is moving west to the position 8 feet along maple street

------------------------------------------
Time = 11:00:07

Rob(Exe)>> Continue current step
Rob(Exe)>> Move along
```

```
Robot requests a look at upcoming street sign

Robot is entering an intersection
Robot is currently in motion...
Robot is moving west to 5 feet along and 3 feet across
the sidewalk at maple and elm


-----------------------------------------
Time = 11:00:08
```

ROBBIE has achieved half of its plan, now it must turn north and move to the final goal location

```
Rob(Exe)>> Step successful!!
Rob(Ind)>> Creating planlet index
Rob(Ind)>> Index is:
(planlet
  (turn (dir north) (on (sidewalk (street elm) (side east))))
  ((facing west)
   (moving #t)
   (mov-speed fast)
   (loc (intersect
          (street1 maple) (side1 north) (block1 100) (along1 5)
          (street2 elm) (side2 east) (block2 100) (along2 3))))
  ())
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case planlet1 matched with difference value of 10
Rob(Ret)>> Case planlet2 matched with difference value of 10
Rob(Ret)>> Cases under consideration are:
  (planlet1 planlet2)
Rob(Exe)>> Choosing between planlets:
(planlet1 planlet2)
Rob(Exe)>> Selecting planlet: planlet2
Rob(Exe)>> Executing a turn to the north on elm

Rob(Exe)>> Come to a stop
Rob(Exe)>> Turning north

Robot is currently stationary...
Robot is facing north at 3 feet along and 5 feet across
the sidewalk at elm and maple

-----------------------------------------
Time = 11:00:09

Rob(CBR)>> Close to entering street south

Rob(Exe)>> Step successful!!
```

```
Rob(Ind)>> Creating planlet index
Rob(Ind)>> Index is:
(planlet
  (move (dir north)
        (on (sidewalk (street elm) (side east)))
        (to (sidewalk (street elm) (side east) (block 200) (along 90))))
  ((facing north)
   (moving #f)
   (mov-speed none)
   (loc (intersect
          (street1 elm) (side1 east) (block1 100) (along1 3)
          (street2 maple) (side2 north) (block2 100) (along2 5))))
  ((cross-street south)))
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case planlet3 matched with difference value of 10
Rob(Ret)>> Case planlet3a matched with difference value of 10
Rob(Ret)>> Case planlet3b matched with difference value of 10
Rob(Ret)>> Case planlet3c matched with difference value of 10
Rob(Ret)>> Case planlet3d matched with difference value of 10
Rob(Ret)>> Cases under consideration are:
  (planlet3 planlet3a planlet3b planlet3c planlet3d)
Rob(Exe)>> Choosing between planlets:
(planlet3 planlet3a planlet3b planlet3c planlet3d)
Rob(Exe)>> Selecting planlet: planlet3d
Rob(Exe)>> Executing a move north on elm

Rob(Exe)>> Move along

Robot is currently in motion...
Robot is moving north to 6 feet along and 5 feet across
the sidewalk at elm and maple

------------------------------------------
Time = 11:00:10

Rob(Exe)>> Continue current step
Rob(Exe)>> Move along

Robot is leaving an intersection
Robot is currently in motion...
Robot is moving north to the position 9 feet along elm street

------------------------------------------
```

... *Execution continues uneventfully until:*

```
------------------------------------------
Time = 11:00:39
```

ROBBIE's robot is approaching Birch street, it must recognize *which street and stop to cross with the green light*

```
Rob(Exe)>> Continue current step
Rob(Exe)>> Move along

Robot requests a look at upcoming street sign

Robot is entering an intersection
Robot is currently in motion...
Robot is moving north to 96 feet along and 5 feet across
the sidewalk at elm and birch

Stoplight at oak and birch changing to yellow.
Stoplight at oak and fir changing to green east and west.
Stoplight at date and birch changing to green north and south.
Stoplight at date and fir changing to green east and west.

------------------------------------------
Time = 11:00:40

Rob(Exe)>> Continue current step
Rob(Exe)>> Move along

Robot is currently in motion...
Robot is moving north to 99 feet along and 5 feet across
the sidewalk at elm and birch

------------------------------------------
Time = 11:00:41

Rob(CBR)>> Close to entering street north

Rob(Exe)>> Context has changed, reconsidering planlet
Rob(Ind)>> Creating planlet index
Rob(Ind)>> Index is:
(planlet
  (move (dir north)
        (on (sidewalk (street elm) (side east)))
        (to (sidewalk (street elm) (side east) (block 200) (along 90))))
  ((facing north)
   (moving #t)
   (mov-speed fast)
   (loc (intersect
          (street1 elm) (side1 east) (block1 100) (along1 99)
          (street2 birch) (side2 south) (block2 100) (along2 5))))
  ((cross-street north)))
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case planlet3 matched with difference value of 10
Rob(Ret)>> Case planlet3a matched with difference value of 10
Rob(Ret)>> Case planlet3b matched with difference value of 10
```

```
Rob(Ret)>> Case planlet3f matched with difference value of 10
Rob(Ret)>> Case planlet3c matched with difference value of 10
Rob(Ret)>> Case planlet3d matched with difference value of 10
Rob(Ret)>> Case planlet3g matched with difference value of 10
Rob(Ret)>> Case planlet3h matched with difference value of 10
Rob(Ret)>> Cases under consideration are:
  (planlet3 planlet3a planlet3b planlet3f planlet3c
   planlet3d planlet3g planlet3h)
Rob(Exe)>> Choosing between planlets:
(planlet3 planlet3a planlet3b planlet3f planlet3c
 planlet3d planlet3g planlet3h)
Rob(Exe)>> Selecting planlet: planlet3g
Rob(Exe)>> Executing a move north on elm

Rob(Exe)>> Storing current step
Rob(Exe)>> Making sub-goal: cross
Rob(Exe)>> Come to a stop
Rob(Exe)>> Looking at stop-light

Robot requests a look at upcoming stoplight

Robot is currently stationary...
Robot is facing north at 99 feet along and 5 feet across
the sidewalk at elm and birch

------------------------------------------
Time = 11:00:42

Rob(CBR)>> Close to entering street north
```

The light is not green, so ROBBIE will stay put until it is

```
Rob(Exe)>> Context has changed, reconsidering planlet
Rob(Ind)>> Creating planlet index
Rob(Ind)>> Index is:
(planlet
  (cross elm birch north)
  ((facing north)
   (moving #f)
   (mov-speed none)
   (loc (intersect
          (street1 elm) (side1 east) (block1 100) (along1 99)
          (street2 birch) (side2 south) (block2 100) (along2 5))))
  ((stop-light e/w) (cross-street north)))
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case planlet5a matched with difference value of 10
Rob(Ret)>> Cases under consideration are:
  (planlet5a)
Rob(Exe)>> Choosing between planlets:
(planlet5a)
```

```
Rob(Exe)>> Selecting planlet: planlet5a
Rob(Exe)>> Executing a street crossing north at elm and birch

Rob(Exe)>> Come to a stop

Robot is currently stationary...
Robot is facing north at 99 feet along and 5 feet across
the sidewalk at elm and birch

------------------------------------------
Time = 11:00:43

Rob(CBR)>> Close to entering street north

Rob(Exe)>> Context has changed, reconsidering planlet
```
Context always changes so long as trying to cross the street!
```
Rob(Ind)>> Creating planlet index
Rob(Ind)>> Index is:
(...)
```
As previous step
```
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case planlet5a matched with difference value of 10
Rob(Ret)>> Cases under consideration are:
  (planlet5a)
Rob(Exe)>> Choosing between planlets:
(planlet5a)
Rob(Exe)>> Selecting planlet: planlet5a
Rob(Exe)>> Executing a street crossing north at elm and birch

Rob(Exe)>> Come to a stop

Robot is currently stationary...
Robot is facing north at 99 feet along and 5 feet across
the sidewalk at elm and birch

------------------------------------------

...
```
Execution continues uneventfully until:
```
------------------------------------------
Time = 11:00:49

Rob(CBR)>> Close to entering street north

Rob(Exe)>> Context has changed, reconsidering planlet
Rob(Ind)>> Creating planlet index
Rob(Ind)>> Index is:
(...)
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case planlet5a matched with difference value of 10
```

```
Rob(Ret)>> Cases under consideration are:
  (planlet5a)
Rob(Exe)>> Choosing between planlets:
(planlet5a)
Rob(Exe)>> Selecting planlet: planlet5a
Rob(Exe)>> Executing a street crossing north at elm and birch

Rob(Exe)>> Come to a stop

Robot is currently stationary...
Robot is facing north at 99 feet along and 5 feet across
the sidewalk at elm and birch

Stoplight at elm and birch changing to green north and south.
Stoplight at cedar and fir changing to green east and west.

-----------------------------------------
Time = 11:00:50

Rob(CBR)>> Close to entering street north
```

The traffic signal changed color in the previous time step,
now ROBBIE can cross the street

```
Rob(Exe)>> Step successful!!
Rob(Ind)>> Creating planlet index
Rob(Ind)>> Index is:
(planlet
  (move (dir north)
        (on (sidewalk (street elm) (side east)))
        (to (sidewalk (street elm) (side east) (block 200) (along 90))))
  ((facing north)
   (moving #f)
   (mov-speed none)
   (loc (intersect
          (street1 elm) (side1 east) (block1 100) (along1 99)
          (street2 birch) (side2 south) (block2 100) (along2 5))))
  ((stop-light n/s) (cross-street north)))
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case planlet3 matched with difference value of 10
Rob(Ret)>> Case planlet3a matched with difference value of 10
Rob(Ret)>> Case planlet3b matched with difference value of 10
Rob(Ret)>> Case planlet3f matched with difference value of 10
Rob(Ret)>> Case planlet3c matched with difference value of 10
Rob(Ret)>> Case planlet3d matched with difference value of 10
Rob(Ret)>> Case planlet3g matched with difference value of 10
Rob(Ret)>> Case planlet3h matched with difference value of 10
Rob(Ret)>> Cases under consideration are:
  (planlet3 planlet3a planlet3b planlet3f planlet3c
   planlet3d planlet3g planlet3h)
```

```
Rob(Exe)>> Choosing between planlets:
(planlet3 planlet3a planlet3b planlet3f planlet3c
 planlet3d planlet3g planlet3h)
Rob(Exe)>> Selecting planlet: planlet3h
Rob(Exe)>> Executing a move north on elm

Rob(Exe)>> Looking away from stop-light
Rob(Exe)>> Move along

Robot is currently in motion...
Robot is moving north to the position 2 feet along
crossing birch on elm

Stoplight at cherry and birch changing to yellow.

------------------------------------------
Time = 11:00:51
```

Because it has moved into the street, it must once again select a new planlet

```
Rob(Exe)>> Context has changed, reconsidering planlet
Rob(Ind)>> Creating planlet index
Rob(Ind)>> Index is:
(planlet
  (move (dir north)
        (on (sidewalk (street elm) (side east)))
        (to (sidewalk (street elm) (side east) (block 200) (along 90))))
  ((facing north)
   (moving #t)
   (mov-speed fast)
   (loc (in-street-inters
          (street1 elm) (side1 east) (block1 (100 200)) (along1 2)
          (street2 birch) (side2 in) (block2 100) (along2 5) (width2 20))))
  ())
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case planlet3 matched with difference value of 10
Rob(Ret)>> Case planlet3a matched with difference value of 10
Rob(Ret)>> Case planlet3b matched with difference value of 10
Rob(Ret)>> Case planlet3c matched with difference value of 10
Rob(Ret)>> Case planlet3d matched with difference value of 10
Rob(Ret)>> Cases under consideration are:
  (planlet3 planlet3a planlet3b planlet3c planlet3d)
Rob(Exe)>> Choosing between planlets:
(planlet3 planlet3a planlet3b planlet3c planlet3d)
Rob(Exe)>> Selecting planlet: planlet3d
Rob(Exe)>> Executing a move north on elm

Rob(Exe)>> Move along
```

```
Robot is currently in motion...
Robot is moving north to the position 5 feet along
crossing birch on elm

------------------------------------------
Time = 11:00:52

Rob(Exe)>> Continue current step
Rob(Exe)>> Move along

Robot is currently in motion...
Robot is moving north to the position 8 feet along
crossing birch on elm

Stoplight at elm and maple changing to yellow.
Stoplight at elm and fir changing to yellow.

------------------------------------------
```

... *Execution continues uneventfully until:*

```
------------------------------------------
Time = 11:00:56

Rob(CBR)>> Close to exiting street north
```

Because it is ready to exit the street, it selects a new planlet

```
Rob(Exe)>> Context has changed, reconsidering planlet
Rob(Ind)>> Creating planlet index
Rob(Ind)>> Index is:
(planlet
  (move (dir north)
        (on (sidewalk (street elm) (side east)))
        (to (sidewalk (street elm) (side east) (block 200) (along 90))))
  ((facing north)
   (moving #t)
   (mov-speed fast)
   (loc (in-street-inters
          (street1 elm) (side1 east) (block1 (100 200)) (along1 17)
          (street2 birch) (side2 in) (block2 100) (along2 5) (width2 20))))
  ((exit-street north)))
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case planlet3 matched with difference value of 10
Rob(Ret)>> Case planlet3a matched with difference value of 10
Rob(Ret)>> Case planlet3b matched with difference value of 10
Rob(Ret)>> Case planlet3c matched with difference value of 10
Rob(Ret)>> Case planlet3d matched with difference value of 10
Rob(Ret)>> Cases under consideration are:
```

```
  (planlet3 planlet3a planlet3b planlet3c planlet3d)
Rob(Exe)>> Choosing between planlets:
(planlet3 planlet3a planlet3b planlet3c planlet3d)
Rob(Exe)>> Selecting planlet: planlet3d
Rob(Exe)>> Executing a move north on elm

Rob(Exe)>> Move along

Robot is currently in motion...
Robot is moving north to the position 20 feet along
crossing birch on elm

------------------------------------------
Time = 11:00:57

Rob(CBR)>> Close to exiting street north

Rob(Exe)>> Context has changed, reconsidering planlet
Rob(Ind)>> Creating planlet index
Rob(Ind)>> Index is:
(planlet
  (move (dir north)
        (on (sidewalk (street elm) (side east)))
        (to (sidewalk (street elm) (side east) (block 200) (along 90))))
  ((facing north)
   (moving #t)
   (mov-speed fast)
   (loc (in-street-inters
          (street1 elm) (side1 east) (block1 (100 200)) (along1 20)
          (street2 birch) (side2 in) (block2 100) (along2 5) (width2 20))))
  ((exit-street north)))
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case planlet3 matched with difference value of 10
Rob(Ret)>> Case planlet3a matched with difference value of 10
Rob(Ret)>> Case planlet3b matched with difference value of 10
Rob(Ret)>> Case planlet3c matched with difference value of 10
Rob(Ret)>> Case planlet3d matched with difference value of 10
Rob(Ret)>> Cases under consideration are:
  (planlet3 planlet3a planlet3b planlet3c planlet3d)
Rob(Exe)>> Choosing between planlets:
(planlet3 planlet3a planlet3b planlet3c planlet3d)
Rob(Exe)>> Selecting planlet: planlet3d
Rob(Exe)>> Executing a move north on elm

Rob(Exe)>> Move along

Robot is currently in motion...
Robot is moving north to 3 feet along and 5 feet across
the sidewalk at elm and birch
```

```
Stoplight at elm and maple changing to green north and south.
Stoplight at elm and fir changing to green east and west.

------------------------------------------
Time = 11:00:58

Rob(CBR)>> Close to entering street south

Rob(Exe)>> Context has changed, reconsidering planlet
Rob(Ind)>> Creating planlet index
Rob(Ind)>> Index is:
(planlet
  (move (dir north)
        (on (sidewalk (street elm) (side east)))
        (to (sidewalk (street elm) (side east) (block 200) (along 90))))
  ((facing north)
   (moving #t)
   (mov-speed fast)
   (loc (intersect
          (street1 elm) (side1 east) (block1 200) (along1 3)
          (street2 birch) (side2 north) (block2 100) (along2 5))))
  ((cross-street south)))
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case planlet3 matched with difference value of 10
Rob(Ret)>> Case planlet3a matched with difference value of 10
Rob(Ret)>> Case planlet3b matched with difference value of 10
Rob(Ret)>> Case planlet3c matched with difference value of 10
Rob(Ret)>> Case planlet3d matched with difference value of 10
Rob(Ret)>> Cases under consideration are:
  (planlet3 planlet3a planlet3b planlet3c planlet3d)
Rob(Exe)>> Choosing between planlets:
(planlet3 planlet3a planlet3b planlet3c planlet3d)
Rob(Exe)>> Selecting planlet: planlet3d
Rob(Exe)>> Executing a move north on elm

Rob(Exe)>> Move along

Robot is currently in motion...
Robot is moving north to 6 feet along and 5 feet across
the sidewalk at elm and birch

------------------------------------------
Time = 11:00:59

Rob(Exe)>> Continue current step
Rob(Exe)>> Move along

Robot is leaving an intersection
Robot is currently in motion...
Robot is moving north to the position 9 feet along elm street
```

```
Stoplight at date and apple changing to green east and west.

------------------------------------------
Time = 11:00:60

Rob(Exe)>> Continue current step
Rob(Exe)>> Move along

Robot is currently in motion...
Robot is moving north to the position 12 feet along elm street

------------------------------------------
```

... *Execution continues uneventfully until:*

```
------------------------------------------
Time = 11:01:26

Rob(Exe)>> Continue current step
Rob(Exe)>> Move along

Robot is currently in motion...
Robot is moving north to the position 90 feet along elm street

------------------------------------------
Time = 11:01:27
```

ROBBIE has reached the goal location, the `move` *step is over and only the ending verification of the location remains*

```
Rob(Exe)>> Step successful!!
Rob(Ind)>> Creating planlet index
Rob(Ind)>> Index is:
(planlet
  (ending-at
    (dir any)
    (on (sidewalk (street elm) (side east) (block 200) (along 90))))
  ((facing north)
   (moving #t)
   (mov-speed fast)
   (loc (sidewalk (street elm) (side east) (block 200) (along 90) (across 5))))
  ())
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case planlet0b matched with difference value of 10
Rob(Ret)>> Case planlet0d matched with difference value of 10
Rob(Ret)>> Cases under consideration are:
  (planlet0b planlet0d)
Rob(Exe)>> Choosing between planlets:
(planlet0b planlet0d)
```

```
Rob(Exe)>> Selecting planlet: planlet0b
Rob(Exe)>> Executing ending-at

Rob(Exe)>> Come to a stop

Robot is currently stationary...
Robot is facing north at the position 90 feet along elm street

Stoplight at elm and fir changing to green north and south.

------------------------------------------
Time = 11:01:28

Rob(Exe)>> Step successful!!
Rob(Exe)>> Plan complete
Rob(Exe)>> Final status:
((facing north)
 (moving #f)
 (mov-speed none)
 (loc (sidewalk (street elm) (side east) (block 200) (along 90) (across 5))))

Robot is currently stationary...
Robot is facing north at the position 90 feet along elm street

------------------------------------------
Time = 11:01:29
```

ROBBIE must store the resulting plan in memory
(no introspective failures occur in this sample run)

```
Rob(Sto)>> Reconstructing finished plan
Rob(Sto)>> Irrelevancy: eliminate street crossing
Rob(Sto)>> Steps need combining
Rob(Sto)>> The plan to be stored:
(#:g1915
  (plan (sidewalk (street maple) (side north) (block 100) (along 20) (across 3))
        (sidewalk (street elm) (side east) (block 200) (along 90)))
  ((starting-at
     (dir any)
     (on (sidewalk (street maple) (side north) (block 100)
                   (along 20) (across 3))))
   (turn (dir west)
         (on (sidewalk (street maple) (side north) (block 100)
                       (along 20) (across 3))))
   (move (dir west)
         (on (sidewalk (street maple) (side north)))
         (to (sidewalk (street elm) (side east))))
   (turn (dir north) (on (sidewalk (street elm) (side east))))
   (move (dir north)
         (on (sidewalk (street elm) (side east)))
         (to (sidewalk (street elm) (side east) (block 200) (along 90))))
```

```
   (ending-at
     (dir any)
     (on (sidewalk (street elm) (side east) (block 200) (along 90))))))

Rob(Ind)>> Creating plan solution index
Rob(Ind)>> Index is:
(solution
  (starting-at
    (dir any)
    (on (sidewalk (street maple) (side north) (block 100)
                  (along 20) (across 3))))
  (turn (dir west)
        (on (sidewalk (street maple) (side north) (block 100)
                      (along 20) (across 3))))
  (move (dir west)
        (on (sidewalk (street maple) (side north)))
        (to (sidewalk (street elm) (side east))))
  (turn (dir north) (on (sidewalk (street elm) (side east))))
  (move (dir north)
        (on (sidewalk (street elm) (side east)))
        (to (sidewalk (street elm) (side east) (block 200) (along 90))))
  (ending-at
    (dir any)
    (on (sidewalk (street elm) (side east) (block 200) (along 90)))))
Rob(Ret)>> Starting to retrieve
Rob(Ret)>> Retrieval level at 3
Rob(Ret)>> Case old1 matched with difference value of 44
Rob(Ret)>> Case old2 matched with difference value of 14
Rob(Ret)>> Case old3 matched with difference value of 36
Rob(Ret)>> Cases under consideration are:
  (old1 old2 old3)
Rob(Sto)>> Storing plan in memory under name: g1915
Rob(CBR)>> ====================================================
Rob(CBR)>> Number of cases considered in retrieval: 3
Rob(CBR)>> Total number of cases in memory: 65
Rob(CBR)>> Number of plans in memory: 4
Rob(CBR)>> Number of rules in memory: 0
Rob(CBR)>> Adaptation was done
Rob(CBR)>> Replanning was done 0 times
Rob(CBR)>> New plan was stored

...
```

Appendix B

Taxonomy of Failures

This appendix contains a listing of the failure taxonomy as it was developed for ROBBIE. The taxonomy was developed by considering the planning process in detail and determining points where failures might occur for each component. Some types of failures are the same in different components because of the re-use of case-based reasoning for parts of the whole planning process.

Trace of reasoning process	Types of failures
Indexer:	Relevant feature missing from criteria
1. Given the initial description,	
2. Retrieve all matching rules	Fails to retrieve relevent rule
using coordinates for locations,	Builds `indexer` index incorrectly
and unification to match indices	Matching picks wrong rule, or misses one
3. For each rule, add new feature to description	Feature incorrectly added
and go on to the next rule	Fails to apply all retrieved rules
4. Pass final description back	Fails to have start, end, or special features in the right form

Trace of reasoning process	Types of failures
Retriever:	Fails to retrieve the best case
	Retrieves wrong number/kind of cases
1. Loop starting with highest restrictiveness of similarity	Tries to retrieve with invalid restrictiveness
2. Make a list of cases whose similarity assessment is good enough	Wrong cases judged similar
	Use wrong similarity measure
	Weighting of similarity stresses wrong features
3. If no matches, decrease restrictiveness and repeat (from 1)	
4. Else if "many" to be retrieved then return all matches,	Retrieves zero matches when at least one required
5. Else select those from the list of equal similarity,	Fails to select most similar
6. Pick randomly from among equally similar	
7. Pass selected case(s) back	Retrieved case has the wrong type
	Wrong number of cases retrieved
	Case retrieved is ill-formed
Adaptor:	Fails to adapt adaptable case
	Does not have a strategy for adapting some situation
	An adaptable case exists in memory, but isn't the retrieved one
1. Map the old cardinal directions onto new (i.e. north⇒north, east⇒west, etc.)	Finds no unique mapping
2. If mapping failed, suggest re-retrieval	Finds no mapping when one exists
3. Else create a template new solution	Skeletal plan is incorrect
5. Set up supplemental information	information incorrectly initialized
6. Repeat until the new case is finished	New case never gets finished
	Misses an unadapted slot
—Select place to adapt	Selects an already adapted place
—Retrieve an applicable strategy	Retrieves irrelevant strategy
	Finds no strategy applicable
—Apply to the hole	Application incorrectly alters plan
7. Return adapted case	Case is incompletely adapted
	Case is incorrectly adapted

Trace of reasoning process	**Types of failures**
Re-retriever:	
1. Select an intermediate location	Chooses bad intermediate location
	Chooses invalid intermediate location (i.e., location does not exist)
2. Retrieve and adapt a case from start to intermediate	(failures for Retriever and Adaptor apply here)
3. Retrieve-and-adapt a case from intermediate to end	
4. Concatenate the cases	Steps are omitted in concatenating
	Extra steps are included in concatenating
5. Return the result	Case has wrong, unusable structure
Executor:	Fails to have a planlet for some situation
	Fails to apply a planlet
1. Given a plan, initialize system for execution	Sets up internal view of world incorrectly
2. While there are plan-steps left	Plan execution never ends
— If starting a step or context has changed	Fails to notice context change
	Fails to update context correctly
— a. Retrieve planlets from memory	Does not retrieve applicable planlet
— b. Pick the planlet matching current context	Picks the wrong planlet
— c. Perform subactions in this timestep	Actions cause a failure
— Else continue current actions	Actions cause a failure
3. When plan complete pass execution log to Storer	Plan does not lead to the goal
	At goal while plan steps still pending
	Collides with an obstacle
	Finds itself in unexpected location
	Takes too long to execute plan
Storer:	Gives the same plan steps two different indices
1. Reconstruct final solution from steps	Fails to reconstruct all plan steps
	Inserts plan steps not in logs
2. Search memory for an exact match,	Stores an identical plan in memory twice
3. If find an exact match, don't store case,	
4. Else add case to memory	Gives new plan bad index
	Stores new plan incorrectly

Appendix C

ROBBIE's Introspective Model

This appendix contains a portion of the actual model structure for ROBBIE. The model structure is a set of clusters containing assertions and connections to each other. Each assertion also contains links to other assertions causally related to it, a list of the underlying information upon which it depends, and a repair class, if any exists. The portions of the model include the cluster for the system as a whole, and the clusters, abstract and specific, for the Indexer component of the planner.

```
(define model
  '((whole
     (parts indexer retriever adaptor re-retriever executor storer)
     (assertions))
    (indexer
     (parts indexer-specific)
     (assertions
      (1 (depend-on-spec "the indexer knows all relevant features")
         (when on-failure)
         ()
         (links (spec (indexer-specific 1))
                (next (indexer 2)))
         (repair))
      (2 (depend-on-spec "the indexer will build the correct index")
```

```
        (when on-failure during)
        ()
        (links (spec (indexer-specific 2))
               (spec (indexer-specific 3))
               (spec (indexer-specific 4))
               (next (indexer 3))
               (prev (indexer 1)))
        (repair))
     (3 (and (depend-on-spec
                "the indexer will produce a valid index")
             (depend-on-next
                "the indexer will produce a valid index"))
        (when on-failure after)
        ()
        (links (spec (indexer-specific 5))
               (spec (indexer-specific 6))
               (spec (indexer-specific 7))
               (next (retriever 1))
               (prev (indexer 2)))
        (repair))))
  (indexer-specific
   (parts)
   (assertions
     (1 (values-compare-by find-features final-solution context)
        (when on-failure)
        (find-features final-solution context)
        (links (abstr (indexer 1))
               (next (indexer-specific 2)))
        (repair add-case))
     (2 (or  (not (has-value index-type plan))
             (has-type goal location?))
        (when before)
        (index-type goal location?)
        (links (abstr (indexer 2))
               (prev (indexer-specific 1))
               (next (indexer-specific 2)))
        (repair))
     (3 (magnitudes-compare-by > new-index index)
        ; the relevant new features will be added
        (when during)
        (new-index index > )
        (links (abstr (indexer 2))
```

```
        (prev (indexer-specific 2))
        (next (indexer-specific 4)))
   (repair))
(4 (or (not (has-value index-type plan))
       (contains-part-of-type index other-part spec-index?))
   ; new features will be added correctly
   (when during)
   (index-type index other-part spec-index?)
   (links (abstr (indexer 2))
        (prev (indexer-specific 3))
        (next (indexer-specific 5))
        (next (indexer-specific 6))
        (next (indexer-specific 7)))
   (repair))

(5 (or (not (has-value index-type plan))
       (contains-part-of-type index from-part location?))
   ; the final index will have a from location
   (when after)
   (index index-type from-part location?)
   (links (abstr (indexer 3))
        (prev (indexer-specific 4)))
   (repair))
(6 (or (not (has-value index-type plan))
       (contains-part-of-type index to-part location?))
   ; the final index will have a to location
   (when after)
   (index index-type to-part location?)
   (links (abstr (indexer 3))
        (prev (indexer-specific 4)))
   (repair))
(7 (or (not (has-value index-type plan))
       (contains-part-of-type index other-part spec-index?))
   ; the final index will have a proper special-part
   (when after)
   (index index-type other-part spec-index?)
   (links (abstr (indexer 3))
        (prev (indexer-specific 4)))
   (repair))))
```

Bibliography

Agre, P. & Chapman, D. (1987). Pengi: an implementation of a theory of activity. In *Proceedings of the Sixth Annual National Conference on Artificial Intelligence* Seattle, WA. AAAI.

Alterman, R. (1986). An adaptive planner. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pp. 65–69 Philadelphia, PA. AAAI.

Arcos, J. & Plaza, E. (1993). A reflective architecture for integrated memory-based learning and reasoning. In Wess, S., Altoff, K., & Richter, M. (Eds.), *Topics in Case-Based Reasoning*. Springer-Verlag, Kaiserslautern, Germany.

Bain, W. (1986). *Case-based Reasoning: A Computer Model of Subjective Assessment*. Ph.D. thesis, Yale University. Computer Science Department Technical Report 470.

Barletta, R. & Mark, W. (1988). Explanation-based indexing of cases. In Kolodner, J. (Ed.), *Proceedings of a Workshop on Case-Based Reasoning*, pp. 50–60 Palo Alto. DARPA, Morgan Kaufmann, Inc.

Bhatta, S. & Goel, A. (1993). Model-based learning of structural indices to design cases. In *Proceedings of the IJCAI-93 Workshop on Reuse of Design* Chambery, France. IJCAI.

Birnbaum, L., Collins, G., Brand, M., Freed, M., Krulwich, B., & Pryor, L. (1991). A model-based approach to the construction of adaptive case-based planning systems. In Bareiss, R. (Ed.), *Proceedings of the Case-Based Reasoning Workshop*, pp. 215–224 San Mateo. DARPA, Morgan Kaufmann, Inc.

Birnbaum, L., Collins, G., Freed, M., & Krulwich, B. (1990). Model-based diagnosis of planning failures. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pp. 318–323 Boston, MA. AAAI.

Brooks, R. (1987). Intelligence without representation. In *Proceedings of the Workshop on the Foundations of Artificial Intelligence* Cambridge, MA. MIT.

Burke, R. (1993). Retrieval strategies for tutorial stories. In Leake, D. (Ed.), *Proceedings of the AAAI-93 Workshop on Case-Based Reasoning*, pp. 118–124 Washington, DC. AAAI. AAAI Press technical report WS-93-01.

Chandrasekaran, B. (1994). Functional representation and causal processes. In Yovits, M. (Ed.), *Advances in Computers*. Academic Press, New York.

Chi, M. & Glaser, R. (1980). The measurement of expertise: a development of knowledge and skill as a basis for assessing achievement. In Baker, E. & Quellmalz, E. (Eds.), *Educational testing and evaluation: Design, analysis and policy*. Sage Publications, Beverly Hill, CA.

Cohen, P. & Howe, A. (1988). How evaluation guides ai research. *The AI Magazine*, *9*(4), 35–43.

Collins, G., Birnbaum, L., Krulwich, B., & Freed, M. (1993). The role of self-models in learning to plan. In *Foundations of Knowledge Acquisition: Machine Learning*, pp. 83–116. Kluwer Academic Publishers.

Cottrell, G. W. & Tsung, F.-S. (1989). Learning simple arithmetic procedures. In *Proceedings of the Eleventh Annual Conference of the Cognitive Science Society*, pp. 58–65 Hillsdale, NJ. Lawrence Erlbaum Associates.

Cox, M. (1994). Machines that forget: learning from retrieval failure of mis-indexed explanations. In *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, pp. 225–230. Lawrence Erlbaum Associates.

Cox, M. (1995). Representing mental events (or the lack thereof). In *Proceedings of the 1995 AAAI Spring Symposium on Representing Mental States and Mechanisms.*

Cox, M. & Freed, M. (Eds.). (1995). *Proceedings of the 1995 AAAI Spring Symposium on Representing Mental States and Mechanisms*, Stanford, CA. AAAI.

DeJong, G. & Mooney, R. (1986). Explanation-based learning: an alternative view. *Machine Learning*, *1*(1), 145–176.

Downs, J. & Reichgelt, H. (1991). Integrating classical and reactive planning within an architecture for autonomous agents. In *European Workshop on Planning*, pp. 13–26.

Efron, B. & Tibshirani, R. (1993). *An Introduction to the Boostrap.* Chapman & Hall.

Elman, J. L. (1991). Incremental learning, or the importance of starting small. *Annual Conference of the Cognitive Science Society*, *13*, 443–448.

Evett, M. (1994). *PARKA: A System for Massively Parallel Knowledge Representation.* Ph.D. thesis, University of Maryland.

Firby, J., Kahn, R., Prokopowicz, P., & Swain, M. (1995). An architecture for vision and action. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pp. 72–79. Morgan Kaufmann Publishers.

Firby, R. J. (1989). *Adaptive Execution in Complex Dynamic Worlds*. Ph.D. thesis, Yale University, Computer Science Department. Technical Report 672.

Flavell, J. (1985). *Cognitive Development*. Prentice-Hall, Englewood Cliffs, NJ.

Flavell, J., Friedrichs, A., & Hoyt, J. (1970). Develpmental changes in memorization processes. *Cognitive Psychology*, *1*, 324–340.

Fowler, N., Cross, S., & Owens, C. (1995). The ARPA-Rome knowledge-based planning initiative. *IEEE Expert*, *10*(1), 4–9.

Fox, S. & Leake, D. (1994). Using introspective reasoning to guide index refinement in case-based reasoning. In *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, pp. 324–329 Atlanta, GA. Lawrence Erlbaum Associates.

Fox, S. & Leake, D. (1995a). An introspective reasoning method for index refinement. In *Proceedings of 14th international Joint Conference on Artificial Intelligence*. IJCAI.

Fox, S. & Leake, D. (1995b). Learning to refine indexing by introspective reasoning. In *Proceedings of the First International Conference on Case-Based Reasoning* Sesimbra, Portugal.

Fox, S. & Leake, D. (1995c). Modeling case-based planning for repairing reasoning failures. In *Proceedings of the 1995 AAAI Spring Symposium on Representing Mental States and Mechanisms* Stanford, CA. AAAI.

Freed, M. & Collins, G. (1994a). Adapting routines to improve task coordination. In *Proceedings of the 1994 Conference on AI Planning Systems*, pp. 255–259.

Freed, M. & Collins, G. (1994b). Learning to prevent task interactions. In desJardins, M. & Ram, A. (Eds.), *Proceedings of the 1994 AAAI Spring Symposium on Goal-driven Learning*, pp. 28–35. AAAI Press.

Gat, E. (1992). Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-worl mobile robots. In *Proceedings, Tenth National Conference on Artificial Intelligence*, pp. 809–815.

Gentner, D. (1988). Metaphor as structure mapping: the relational shift. *Child Development*, *59*, 47–59.

Goel, A., Ali, K., & de Silva Garza, A. G. (1994). Computational tradeoffs in experience-based reasoning. In *Proceedings of the AAAI-94 workshop on Case-Based Reasoning*, pp. 55–61 Seattle, WA.

Goel, A., Callantine, T., Shankar, M., & Chandrasekaran, B. (1991). Representation, organization, and use of topographic models of physical spaces for route planning. In *Proceedings of the Seventh IEEE Conference on AI Applications*, pp. 308–314. IEEE Computer Society Press.

Goel, A. & Chandrasekaran, B. (1989). Use of device models in adaptation of design cases. In Hammond, K. (Ed.), *Proceedings of the Case-Based Reasoning Workshop*, pp. 100–109 San Mateo. DARPA, Morgan Kaufmann, Inc.

Gruber, T. (1989). Automated knowledge acquisition for strategic knowledge. *Machine Learning*, *4*, 293–336.

Hammond, C. (1989). *Case-Based Planning: Viewing Planning as a Memory Task*. Academic Press, San Diego.

Hennessey, D. & Hinkle, D. (1991). Initial results from clavier: a case-based autoclave loading assistant. In Bareiss, R. (Ed.), *Proceedings of the Case-Based Reasoning Workshop*, pp. 225–232 San Mateo. DARPA, Morgan Kaufmann, Inc.

Hinrichs, T. (1992). *Problem Solving in Open Worlds: A Case Study in Design*. Lawrence Erlbaum Associates, Hillsdale, NJ.

Ibrahim, M. (1992). Reflection in object-oriented programming. *International Journal on Artificial Intelligence Tools*, *1*(1), 117–136.

Kitano, H., Shibata, A., Shimazu, H., Kajihara, J., & Sato, A. (1992). Building large-scale and corporate-wide case-based systems: integration of organizational and machine executable algorithms. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 843–849 San Jose, CA. AAAI.

Kodratoff, Y. & Michalski, R. (Eds.). (1990). *Machine Learning: An Artificial Intelligence Approach*, Vol. 3. Morgan Kaufmann.

Kolodner, J. (1984). *Retrieval and Organizational Strategies in Conceptual Memory*. Lawrence Erlbaum Associates, Hillsdale, NJ.

Kolodner, J. (1988). Retrieving events from a case memory: a parallel implementation. In Kolodner, J. (Ed.), *Proceedings of a Workshop on Case-Based Reasoning*, pp. 233–249 Palo Alto. DARPA, Morgan Kaufmann, Inc.

Kolodner, J. (1993a). *Case-Based Reasoning*, chap. 8, pp. 289–320. Morgan Kaufman, San Mateo, CA.

Kolodner, J. (1993b). *Case-Based Reasoning*. Morgan Kaufman, San Mateo, CA.

Koton, P. (1989). Smartplan: a case-based resource allocation and scheduling system. In Hammond, K. (Ed.), *Proceedings of the Case-Based Reasoning Workshop*, pp. 290–294 San Mateo. DARPA, Morgan Kaufmann, Inc.

Kreutzer, M., Leonard, M., & Flavell, J. (1975). An interview study of children's knowledge about memory. *Monographs of the Society for Research in Child Development*, *40*. (1, Serial No. 159).

Krulwich, B., Birnbaum, L., & Collins, G. (1992). Learning several lessons from one experience. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pp. 242–247 Bloomington, IN. Cognitive Science Society.

Leake, D. (1992). *Evaluating Explanations: A Content Theory*. Lawrence Erlbaum Associates, Hillsdale, NJ.

Leake, D. (1995). Combining rules and cases to learn case adaptation. In *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society* Pittsburgh, PA. Lawrence Erlbaum Associates.

Leake, D. & Owens, C. (1986). Organizing memory for explanation. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, pp. 710–715 Amherst, MA. Cognitive Science Society.

Lehnert, W. (1978). *The Process of Question Answering*. Lawrence Erlbaum Associates, Hillsdale, NJ.

Machine Learning (1989) *Machine Learning*, *4*(3/4). Special issue on knowledge acquisition.

McDermott, D. (1981). Artificial intelligence meets natural stupidity. In Haugeland, J. (Ed.), *Mind Design*. MIT Press, Bradford Books, Cambridge, MA.

Meeden, L. (1994). *Towards Planning: Incremental Investigations into Adaptive Robot Control*. Ph.D. thesis, Indiana University, Computer Science Department.

Michalski, R., Carbonell, J., & Mitchell, T. (Eds.). (1983). *Machine Learning: An Artificial Intelligence Approach*, Vol. 1. Morgan Kaufmann.

Michalski, R., Carbonell, J., & Mitchell, T. (Eds.). (1986). *Machine Learning: An Artificial Intelligence Approach*, Vol. 2. Morgan Kaufmann.

Michalski, R. & Tecuci, G. (Eds.). (1994). *Machine Learning: A multistrategy approach*, Vol. 4. Morgan Kaufmann.

Minton, S. (1990). Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, *42*, 363–391.

Mitchell, T., Keller, R., & Kedar-Cabelli, S. (1986). Explanation-based generalization: a unifying view. *Machine Learning*, *1*(1), 47–80.

Nourbakhsh, I., Powers, R., & Birchfield, S. (1995). Dervish: an office-navigating robot. *AI Magazine*, *16*(2), 53–60.

Oehlmann, R., Edwards, P., & Sleeman, D. (1995). Introspection planning: representing metacognitive experience. In *Proceedings of the 1995 AAAI Spring Symposium on Representing Mental States and Mechanisms*.

Ram, A. (1989). *Question-driven understanding: An integrated theory of story understanding, memory and learning.* Ph.D. thesis, Yale University, New Haven, CT. Computer Science Department Technical Report 710.

Ram, A. (1993). Indexing, elaboration and refinement: incremental learning of explanatory cases. *Machine Learning*, *10*(3), 201–248.

Ram, A. & Cox, M. (1994). Introspective reasoning using meta-explanations for multistrategy learning. In Michalski, R. & Tecuci, G. (Eds.), *Machine Learning: A Multistrategy Approach.* Morgan Kaufmann.

Redmond, M. (1992). *Learning by Observing and Understanding Expert Problem Solving.* Ph.D. thesis, College of Computing, Georgia Institute of Technology. Technical report GIT-CC-92/43.

Riesbeck, C. (1981). Failure-driven reminding for incremental learning. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pp. 115–120 Vancouver, B.C. IJCAI.

Rosenbloom, P., Laird, J., & Newell, A. (1993a). *Meta Levels in Soar*, Vol. I, chap. 26. The MIT Press.

Rosenbloom, P., Laird, J., & Newell, A. (1993b). *R1-SOAR: An Experiment in Knowledge-Intensive Programming in a Problem Solving Architecture*, Vol. I, chap. 9. The MIT Press.

Ross, B. (1989). Some psychological results on case-based reasoning. In Hammond, K. (Ed.), *Proceedings of the Case-Based Reasoning Workshop*, pp. 144–147 San Mateo. DARPA, Morgan Kaufmann, Inc.

Schank, R. (1982). *Dynamic memory: A theory of learning in computers and people.* Cambridge University Press.

Schank, R. (1986). *Explanation Patterns: Understanding Mechanically and Creatively.* Lawrence Erlbaum Associates, Hillsdale, NJ.

Schank, R. & Leake, D. (1989). Creativity and learning in a case-based explainer. *Artificial Intelligence*, *40*(1-3), 353–385. Also in Carbonell, J., editor, *Machine Learning: Paradigms and Methods*, MIT Press, Cambridge, MA, 1990.

Simoudis, E. (1992). Using case-based retrieval for customer technical support. *IEEE Expert*, *7*(5), 7–13.

Simoudis, E. & Miller, J. (1991). The application of cbr to help desk applications. In Bareiss, R. (Ed.), *Proceedings of the Case-Based Reasoning Workshop*, pp. 25–36 San Mateo. DARPA, Morgan Kaufmann, Inc.

Simpson, R. (1985). *A Computer Model of Case-based Reasoning in Problem-solving: An Investigation in the Domain of Dispute Mediation.* Ph.D. thesis, School of Information and Computer Science, Georgia Institute of Technology. Georgia Institute of Technology, Technical Report GIT-ICS-85/18.

Stanfill, C. & Waltz, D. (1988). The memory-based reasoning paradigm. In Kolodner, J. (Ed.), *Proceedings of a Workshop on Case-Based Reasoning*, pp. 414–424 San Mateo. DARPA, Morgan Kaufmann, Inc.

Stroulia, E. & Goel, A. (1992). Generic teleological mechanisms and their use in case adaptation. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pp. 319–324 Bloomington, IN. Cognitive Science Society.

Stroulia, E. & Goel, A. (1994). Task structures: what to learn?. In desJardins, M. & Ram, A. (Eds.), *Proceedings of the 1994 AAAI Spring Symposium on Goal-driven Learning*, pp. 112–121. AAAI Press.

Sycara, K. & Navinchandra, D. (1989). Index transformation and generation for case retrieval. In Hammond, K. (Ed.), *Proceedings of the Case-Based Reasoning Workshop*, pp. 324–328 San Mateo. DARPA, Morgan Kaufmann, Inc.

Veloso, M. & Carbonell, J. (1993). Derivational analogy in prodigy: automating case acquisition, storage, and utilization. *Machine Learning*, *10*(3), 249–278.