Self-Improvement through Self-Understanding

Model-Based Reflection for Agent Adaptation

A Thesis
Presented to
The Academic Faculty

by

J. William Murdock

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Georgia Institute of Technology
July 2001

Self-Improvement through Self-Understanding

Model-Based Reflection for Agent Adaptation

Approved:

Ashok Goel, Chairman

Alex Kirlik

Sven Koenig

Colin Potts

Ashwin Ram

Spencer Rugaber

Date Approved

Acknowledgements

An enormous number of people have made major contributions to my graduate studies. Foremost among these is my advisor, Ashok Goel, who has provided tremendous insight and guidance throughout every stage of this process. I have been amazingly fortunate to have an advisor who has been so thoroughly involved in the research and who, at the same time, has allowed me the freedom to explore those aspects of the work which I found particularly fascinating. I could not have asked for a better co-pilot on this journey through the study of intelligence and adaptation.

I have also received excellent guidance and support from the other members of my thesis committee: Alex Kirlik, Sven Koenig, Colin Potts, Ashwin Ram, and Spencer Rugaber. Many additional members of the faculty and staff of Georgia Tech have also contributed significantly. Among these are the faculty members with whom I have collaborated in research, including the aforementioned members of my committee and also Gregory Abowd, Mark Guzdial, Michael McCracken, Melody Moore, Sham Navathe, Edward Omiecinski, and Linda Wills. There are also many researchers outside of Georgia Tech who have had a considerable influence on this work, especially David Aha, Dean Allemang, B. Chandrasekaran, and Ram Sriram.

I would also like to acknowledge the contributions of the many past and present Georgia Tech graduate students with whom I have interacted during the course of my graduate studies. I would particularly like to recognize the outstanding contributions of the three fellow students whose work has been most closely related and interconnected with my own: Jim Davies, Todd Griffith, and Eleni Stroulia. I would also like to thank a number of other students who have also played a substantial role: Sam Bhatta, Helene Brashear, Jeff Donahoo, Anthony Francis, David Furcy, Gordon Shippey, Marin Simina, Marty Geier, and Patrick Yaner.

Lastly, I would like to thank the members of my immediate family, Charles, Heather, Hunter, Jim, and Kit Murdock, as well as all of the members of my extended family. Without the support of my family, this dissertation would certainly not be possible.

Contents

List of Tables

List of Figures

Preface

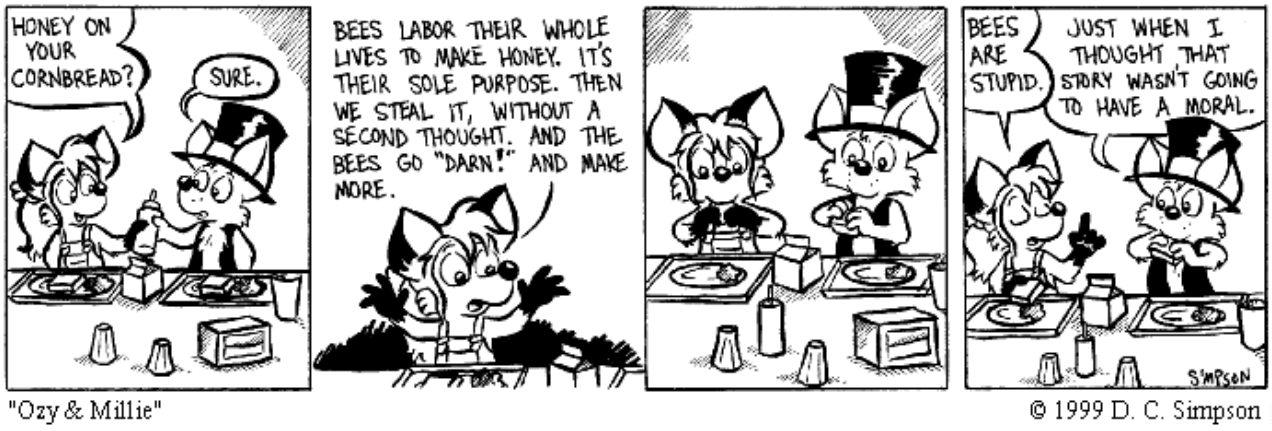
The ability to adapt is a key characteristic of intelligence. This dissertation explores techniques for enabling intelligent software agents to adapt themselves as their functional requirements change incrementally. In the domain of manufacturing, for example, a software agent designed to disassemble physical artifacts may be given a new goal of assembling artifacts. As another example, in the internet domain, a software agent designed to browse some types of documents may be called upon to browse a document of another type.

This research examines the use of reflection (an agent's knowledge and reasoning about itself) to accomplish adaptation (incremental revision of an agent's capabilities). Reflection in this work is enabled by a language called TMKL (Task-Method-Knowledge Language) that supports modeling of an agent's composition and functioning. A TMKL model of an agent explicitly represents the tasks the agent addresses, the methods it applies, and the knowledge it uses. TMKL models are hierarchical, i.e., they represents tasks, methods and knowledge at multiple levels of abstraction. These models are used in a reasoning shell called REM (Reflective Evolutionary Mind). REM provides support for the execution and adaptation of agents which contain TMKL models of themselves.

This dissertation presents a variety of strategies for adapting agents. Some of these strategies are purely model-based: knowledge of composition and functioning directly enables adaptation. This model-based approach is combined with two traditional artificial intelligence and machine learning techniques: generative planning and reinforcement learning. The combination of model-based adaptation, generative planning, and reinforcement learning constitutes a mechanism for reflective agent adaptation which is capable of addressing a variety of problems to which none of these individual approaches alone is suited. The work described in this dissertation has demonstrated the computational feasibility of this mechanism using experiments involving a variety of intelligent software agents in a variety of domains.

Chapter 1

Introduction



(This and other Ozy and Millie comics in this reformatted edition of this thesis were found at http://www.ozyandmillie.org/**. They are all by David C. Simpson. Explicit permission is given on the site for non-commercial redistribution of these strips, as is done here.)**

Flexibility is a key characteristic of intelligence. An intelligent agent with some set of capabilities ought to be able to extend those capabilities to respond to new demands from the environment. In the domain of manufacturing, for example, a software agent designed to disassemble physical artifacts may be asked to instead assemble an artifact. As another example, in the internet domain, a software agent designed to browse some types of documents may be called upon to browse a document of an unknown type. As a third example, in the domain of meeting scheduling, a system may fail to schedule a meeting because it has constraints which specify a range of possible meeting times and the user wants a meeting outside of that range. In all of these cases, the agent needs to change what it does and how it does it.

Some software systems do contain specialized code for making specific changes like the ones in these examples. A programmer can spend an arbitrarily large amount of effort building a system to address an arbitrarily large variety of situations. This process can be extremely expensive, and if the environment is sufficiently dynamic, there will generally still be additional situations which the programmer did not anticipate. A broadly applicable approach based on autonomous self-adaptation could potentially have enormous impact on the way software agents are constructed. The developer of an intelligent software agent could simply create a few simple processes and provide a declarative representation of those processes and the domain that they operate in. As the requirements of the agent evolve the agent may adapt itself to meet the new requirements. The developer would not need to try to build every conceivably useful variation on these processes.

This dissertation describes how an agent can modify itself to meet new challenges by reflecting upon explicit models of its own reasoning and knowledge. The primary claim made here is that model-based adaptation provides an effective balance between knowledge requirements, computational costs, and flexibility; that is, the knowledge requirements of the technique are realistic, the computational costs relative to other methods are small, and yet the technique provides a significant range of adaptive capabilities. Experimental evidence is provided to support this claim.

1.1 Introductory Example

Consider an agent which installs a light bulb via a process with two pieces: inserting the bulb and rotating the bulb. There are many ways that this process might be encoded in a computer program. The most obvious is to write code for the two pieces in a traditional programming language and then to create a main program which just runs the two of them in order. Such a system would probably be very fast but it would not be very flexible; it could not do anything other than install bulbs.

The field of Artificial Intelligence provides an alternative to this approach: reasoning about what to do. There is a broad range of techniques which can address problems of this sort using a variety of kinds of knowledge and processing. Some existing AI techniques which could be used in this situation include reinforcement learning, generative planning, and case-based planning. This dissertation presents an alternative to these approaches: model-based adaptation. Model-based adaptation uses different kinds of knowledge than these other approaches and consequently is able to address problems to which competing approaches are poorly suited.

An agent which addressed the light bulb installation problem through reinforcement learning would require knowledge about what the actions and states in the domain are and what states are desirable. The actions would include inserting and rotating and might also include other kinds of actions one might want to do in the domain of light bulbs such as retracting; the states would include the location of the bulb, and the one desired state would be that the bulb was in the socket. Using this information, the agent would begin trying arbitrary actions and would, over time, learn from experience which actions lead to the desired state. The primary advantage that this agent would have over the traditional software system would be its flexibility. For example, if the goal of the agent was changed to involve removing a bulb instead of a inserting a bulb, the reinforcement learning agent would simply need to be given the new desired state and it could start learning which actions achieve that new state. The drawback to this approach is that it can be very expensive. If the number of actions and states is even moderately large, the amount of experience to learn an adequate action policy can be enormous. Furthermore, the benefits of learning do not extend from one goal to another; when the agent gets a new requirement like removing a bulb, it needs to go through the entire expensive learning process again.

If the agent has additional knowledge, it may be able to decide in advance what actions to take rather than trying to learn through experience. For example, if all of the actions contain logical assertions stating what must be true before and after that action is performed, then an agent may be able to deduce that the combination of the effects of some sequence of actions leads to the desired result; this process is called generative planning. Avoiding the necessity for experience is very valuable. However, generative planning can be still be very time consuming for moderately complex problems. Like reinforcement learning, it needs to start from the beginning each time a new problem is received.

It is possible to reuse entire processes for different goals. For example, case-based planning is a technique in which plans are modified to fit new goals. An case-based planning agent can be seen as having a rudimentary understanding of the process that it goes through; one can view the goals associated with a plan in a case-based planner as being a sort of "task" that the agent needs to perform and the plan that is produced as a sort of "method" for addressing that task. However, this view of tasks and methods is fairly limited. Tasks in this view only involve one specific situation; for example, there could be a task which inserts a specific bulb into a specific socket but there would not be a task which inserts an arbitrary bulb into an arbitrary socket. Furthermore, each task in this paradigm can have only one method, and each method must consist only of a single sequence of actions with no branches or loops. Because the plan specifies exactly what to do, there is no need to explicitly represent any knowledge or reasoning within a plan.

Such an approach is fine for problems which are adequately addressed by rigid sequences of actions without any intervening reasoning. However, other sorts of problems demand more elaborate sorts of tasks and methods, including capabilities like parameterization of the overall task, availability of multiple methods for a task and multiple paths through a method, etc. Note that moving through such a combination of tasks and methods involves making some decisions about what to do; these decisions need to be guided by knowledge within the agent and reasoning about that knowledge. For example, if a task has multiple methods, there must be a specification of the conditions under which each may be applicable, and some inference may be necessary to determine which conditions hold. An explicit representation of tasks, methods, and knowledge of this sort would not merely be a plan for a particular situation but rather would be a *model* of a process which is applicable to a range of situations. Such a model could be used directly to address problems within that range. Furthermore, it would be possible for an agent to extend that range by modifying its methods to address other tasks. Thus the agent is able to behave flexibly by adapting its model of itself to accomplish new results. This approach, reasoning by adaptation of self-models, is the focus of

this dissertation.

Figure 1 shows the adaptation of the light bulb agent using a model. The first portion of this figure provides a diagram of an agent which installs a light bulb. This diagram presents some, but not all, of the information which would be encoded in a model of this agent. The diagram shows that there are three tasks and one method. Furthermore it depicts relationships among them, showing that the main task (Install Light Bulb) is accomplished by the method (Traditional Installation) which involves first invoking one lower level task (Insert Bulb) and then the other (Rotate Bulb). A model of the agent would provide a formal specification of this information, and would provide additional information about these tasks and method, and would also provide information about the knowledge that the agent uses. For example, the description of the Insert Bulb task might contain a formal statement that the result of the task is that a given bulb is located in a given socket.

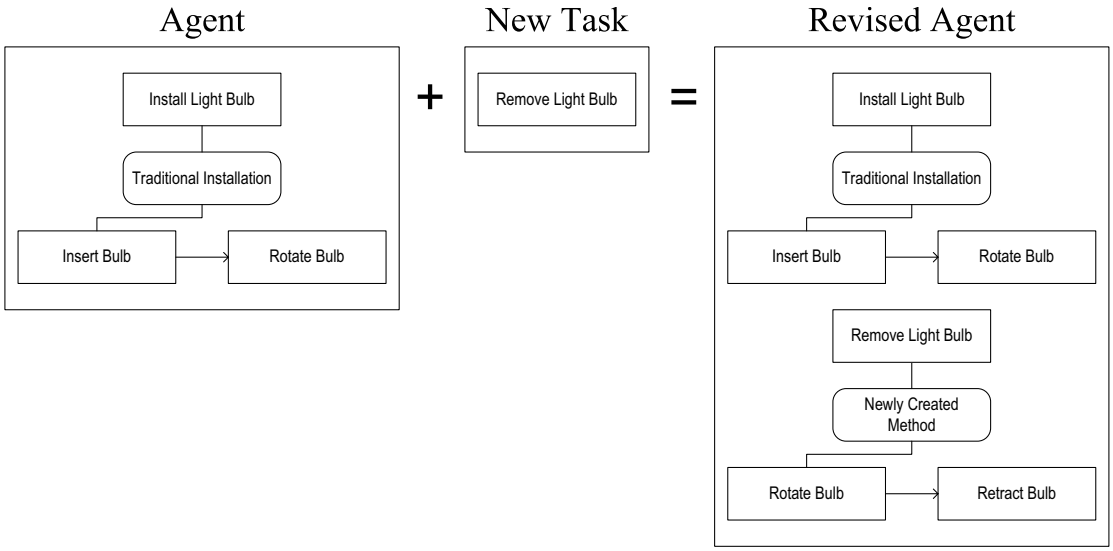


Figure 1: A very simple example of agent adaptation. Small rectangular tags represent tasks (e.g., Install Light Bulb). Small rounded tags represent methods (e.g., Traditional Installation). This notation is used for diagrams of tasks and methods throughout this dissertation.

The second part of Figure 1 shows a new task being requested by the user. The description of the task includes a statement of the result (i.e., that the bulb is out of the socket). It is possible to use this information along with the existing model's encoding of knowledge of bulbs, sockets, locations, etc. to infer that there is similarity in the effects of the new task and the Install Light Bulb task. Consequently, it is possible to adapt the method which accomplishes the existing task so that it is suited to addressing the new task. The result is shown in the last part of Figure 1; a modified agent is provided which is capable of executing both tasks.

1.2 Meta-Reasoning

It is useful to distinguish between two different types of situations in which an agent may need to reflect on its own reasoning. One of these situations is after some execution process in which the desired results were not produced. If an agent has been given some task to perform and it already knows some way of attempting to perform that task *but* that attempt does not do exactly what the user wanted, the agent needs to change in order to perform that task in that environment. For example, if a scheduling agent is asked to schedule a meeting that fits into a given set of times, and no adequate result is found, then the agent cannot satisfy the request unless it makes some change to itself. There are several key questions which an agent needs ask about itself in this circumstance:

- What are the differences between the intended effect and the actual effect?
- What elements of the agent contributed to those differences?
- What modifications can be made to the agent to eliminate those differences?

These are all challenging questions. Computing the difference between intended and actual effects is challenging because intended effects are generally abstract and may be ill-defined. While its typically easy to determine what did happen (at least within the agent), it is not always clear precisely what should have happened. Some knowledge about the overall purpose of the agent can be useful in making this determination; additional feedback from the user may also be needed to clarify what results were desired. Determining which elements contributed to the differences can be challenging in that an agent is likely to have many parts and the contribution that a particular part made to the ultimate result may be complex. Thus localization involves a search over many items in which recognition of the goal is potentially difficult. To address the issue of the quantity of elements, it is helpful to have a record of what happened within the agent during execution (i.e., a trace of the agent's reasoning) to guide the search. To address the issue of recognizing a candidate for change, it is helpful to have knowledge about how pieces of an agent contribute to the overall effect of that agent. Lastly, determining what modifications to make to an element once it is found can be particularly challenging because a piece of a computation can have such a variety of direct and indirect effects in a variety of circumstances (making it very difficult to find a specific modification which accomplishes a desired result). This issue can be addressed using a variety of specific types of transformations to accomplish specific types of differences in behavior; such transformations can be guided by knowledge about these behaviors.

A different type of situation in which reflection may be needed is one in which a novel task is provided. If an agent is asked to perform a task for which it has no existing strategy, it needs to modify itself before it can even begin. The light bulb example in Section 1.1 is one such situation. Modification which occurs before any execution has been performed also raises some difficult questions:

- Is there a known task which accomplishes a similar effect to the new task?
- If so, what are the differences between the effects of the two tasks?
- What elements of the agent contribute to those differences?
- What modifications can be made to the agent to eliminate those differences?

Note that the last three of these questions are very similar to the questions which arose for adaptation after execution. However, addressing these questions is even harder in this, the second type of situation. Finding elements which contribute to the difference between the existing process and the desired one is particularly challenging in this type of situation; since adaptation occurs before any execution has taken place, there is no execution trace to guide the search. Furthermore, there is no specific information about particular pieces of knowledge produced by the tasks; thus modification can not be guided by specific values involved in a particular execution. An agent performing adaptation of this sort can, however, benefit from abstract knowledge about the kinds of knowledge used in addressing the task. For example, if the agent knows that the assembly and disassembly tasks involve states of a device and has additional knowledge about the nature of these states (specifically, that they are the inverse of each other), then this knowledge can be used to suggest what sorts of changes need to be made to the agent (i.e., that the process needs to be inverted).

To summarize, the union of the challenges posed by adaptation in response to failures and adaptation in response to new tasks suggest a variety of desirable characteristics for an agent's representation of itself:

1. What the pieces of the agent do.
2. What the intended effect of the agent is.
3. How the pieces of the agent are combined to accomplish its intended effect (i.e., the connection between 1 and 2).
4. What sorts of information the agent processes.

The TMK (Task-Method-Knowledge) family of agent models has these characteristics. TMK models provide information about the function of systems and their elements (i.e., the tasks that they address) and the behavior of those systems and elements (i.e., the methods that they use) using explicit representations of the information that these elements process (i.e., the knowledge that they apply). Tasks and methods are arranged hierarchically: a high level task (which involves the overall effect of the system) is related to one or more methods (which combine the pieces of the agent into that effect) which are, in turn, broken down into lower level tasks (distinct pieces of the functionality). The explicit representation of the kinds of knowledge that the agent has provides a foundation for

describing how the tasks and methods affect that knowledge. One aspect of the research presented here has been the development of a new formalism for TMK models called TMKL (for TMK Language). TMKL fits within the general framework of TMK, but provides more power and flexibility than previously existing formalizations of TMK.

There are many ways to adapt models of this sort. One approach is the use of a collection of specialized model adaptation strategies. The models provide a basis for analyzing the differences between an existing task and a new, desired task. When a particular type of difference is observed, an appropriate strategy is invoked. TMK models provide extensive information about the overall effects of an agent (i.e., the high level tasks), the activities which accomplish those effects (i.e., the low level tasks), and the connections between them (i.e., the methods). Consequently, this approach is particularly well suited to identifying aspects of the system which relate to the differences between the tasks and can also be useful in identifying the types of changes which need to be made. This approach to adaptation is central to the SIRRINE (Self-Improving Reflective Reasoner Integrating Noteworthy Experience) reasoning shell, which is one of the systems which has been developed for this dissertation research.

One major drawback of purely model-based adaptation, however, is that any fixed strategy is likely to be fairly restricted. Consider an adaptation strategy which simply adds an entry to a binary table for a known relation. Such a strategy may have several distinct varieties of situations in which it is used and corresponding ways in which it needs to behave; for example, if the relation is known to be one-to-one, then the strategy needs to not only add the new fact but also remove any old facts linking the involved concepts in that relation. Because any given knowledge representation formalism only has a limited number of types of relations, it is possible to simply enumerate them and handle each one explicitly. In contrast, an adaptation strategy which makes a dramatic change to an agent across many different aspects of its knowledge and reasoning needs to be able to address the entire variety of knowledge and reasoning that an agent might possess. If the modeling language used for the agents is powerful enough to effectively describe a broad and interesting class of agents, it is overwhelmingly impractical to construct a rigid adaptation strategy that can produce a dramatic change *and* covers all such agents. What is really needed is a set of *flexible* adaptation strategies.

Unfortunately, this line of reasoning seems to suggest that we have reduced the problem of providing flexible reasoning to the problem of providing flexible reasoning! However, this reduction may not be as hopeless as it seems. In particular, many problems which are too large and complex to be addressed by general-purpose reasoning mechanisms can be addressed by relatively simple specialized software. In some situations it may be possible for general-purpose mechanisms to address the modification of the (simple) software systems, even if the (complex) problems that those systems address are beyond the scope of those generic mechanisms. Even when this is not possible, specialized model-based adaptation strategies may be able to quickly and easily accomplish parts of the adaptation process (even if they are not able to handle all aspects of a major adaptation effort). By partially solving an adaptation problem, these strategies may be able to reduce the size of the remaining issues to a level at which general purpose approaches such as generative planning and reinforcement learning may be able to easily address them. In this way it is possible for the combination of model-based adaptation strategies and general-purpose planning and learning mechanisms to work together to effectively solve problems which may be impossible or overwhelmingly expensive for any of these approaches to address by itself. This synergy provides the theoretical foundations of a reasoning shell called REM (Reflective Evolutionary Mind). SIRRINE and REM together represent the implementations of the overall theory presented in this dissertation; experiments with SIRRINE and REM thus form the basis for the evaluation of this theory.

1.3 Meta-Reasoning Architecture

REM and SIRRINE are reasoning shells which provide support for execution and adaptation of intelligent agents. They take as input a collection of tasks and methods (including an explicit statement of what the main task to be addressed is) plus some set of input values. Their overall purpose is to accomplish the main task given the input values; for example, in the domain of engineering a task might be assembly and a set of inputs might include a specific device to assemble. Because an agent within REM or SIRRINE has a model of how it reasons, it can use this model to make changes for itself as needed. The reasoning shells provide the basic mechanisms for executing and modifying an agent.

Figure 2 shows a particularly simple process using TMKL.[1] A task with some method specified for it is simply executed and it produces some output values. One would expect that for many agents in many domains, this sort of behavior would be the most common; i.e., in most situations an agent should already know how to correctly perform the desired task.

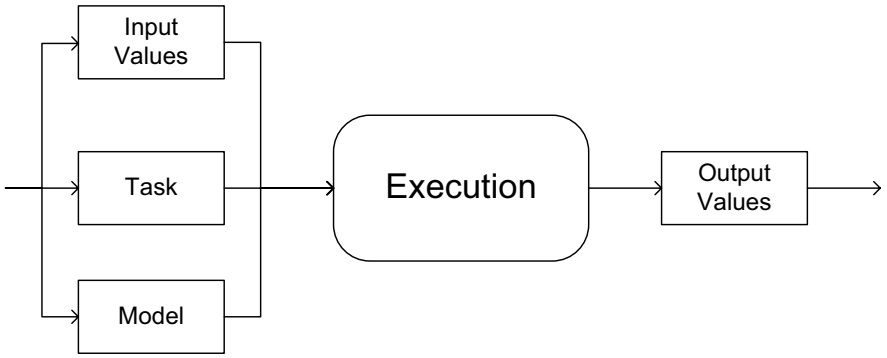


Figure 2: A simple application of TMK models; an implemented task is executed.

Of course, the primary benefits of using models of agents come from exceptional situations, i.e., ones for which the agent needs to adapt. Figure 3 shows one process involving adaptation. It begins like the previous process: an agent is executed with a given task and a given set of inputs. In addition to the outputs, the execution process also provides a trace of reasoning, i.e., a record of which tasks and methods were executed and what intermediate results were produced. A trace differs from a model in that a model can potentially have loops, branches, etc. In contrast, a trace includes only a single path through the process: the one actually taken. If the execution of the agent is successful then the process is completed (as in Figure 2). However, if the execution is does not produce the desired results, some adaptation is required. The trace provides a considerable advantage in performing adaptation in that each step and result which was produced is recorded so a search of the trace can localize portions of the process which made particular contributions to the results. This localization can be used to make changes to the corresponding elements of the TMKL model so that future executions which encounter similar situations use the modified process. The primary task is then executed again using the modified tasks and methods. This approach to agent adaptation and execution is the one taken by SIRRINE.

Figure 4 shows another approach to reasoning with models. In this process, an agent is given a task for which it does not already have any methods. In this case, it must perform some adaptation (to build a method) before any execution can be performed. This adaptation may involve retrieving other, similar tasks from the existing model and modifying the methods for them; for an example of that situation, see Section 1.1. Alternatively, if an existing model is not available, adaptation can be performed using knowledge of the primitive actions in the domain instead. Note that adaptation before execution has a substantial disadvantage versus adaptation after execution: in the former, no trace is available. Consequently, the adaptation strategies must rely only on abstract process and domain information and cannot use a concrete example. Thus adaptation involves reasoning not about how an agent did perform in a particular situation but rather about how an agent could perform in any situation. When an existing method for the required task is available, it is generally beneficial to execute that method and have a trace available to support adaptation (if necessary); however if no method is available, then this initial execution is not possible and adaptation must take place without a trace.

Figure 5 shows the overall reasoning process of REM. This combines the two previous approaches. The inputs to this process are a model with a particular task designated as the main task, plus a set of input values. The intended effect of the process is the accomplishment of the result specified in the description of the main task; an additional effect of the process is that the model has been adapted if necessary. There are two paths through the the process. In the first path, when an implemented task is requested, the task is executed first and then adapted only if necessary.

[1] In this figure, and the ones that follow it in this chapter, the following notations are used. Small rectangular boxes represent pieces of knowledge. Large rounded boxes represent complex processes (which are generally decomposed in later figures in this or other chapters). Smaller rounded boxes represent simpler processes (which are not further broken down in later figures). Arrows that go in from the left and lead to knowledge items indicate that those knowledge items are inputs to the overall process; multiple sets of incoming arrows indicate multiple kinds of uses. Arrows that go from knowledge items out to the right indicate outputs to the overall process. Arrows from knowledge to processes indicate that the knowledge is input for the process. Arrows from processes to knowledge indicate that the knowledge is output from the process.

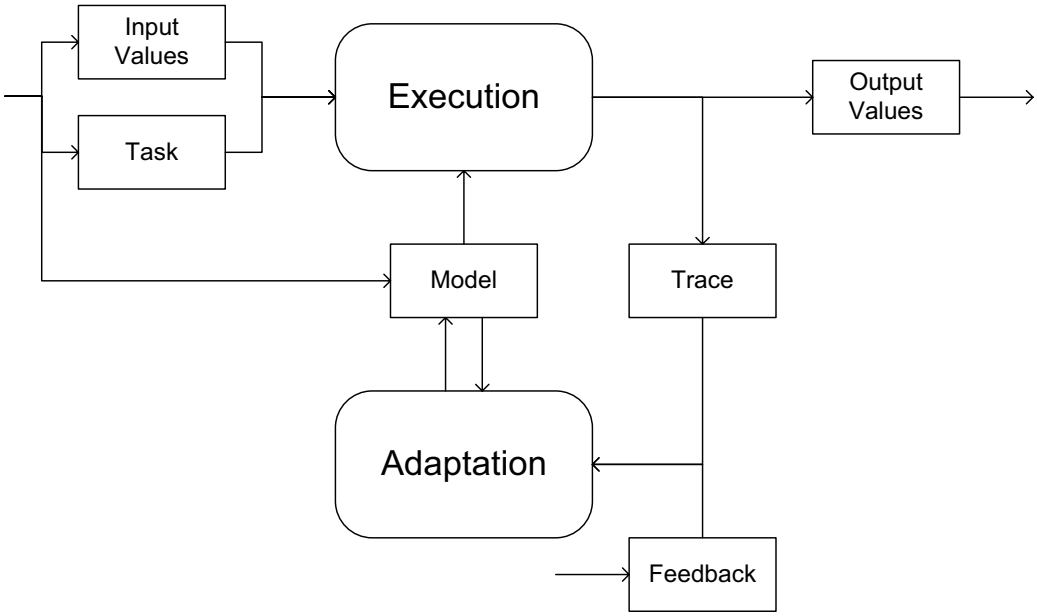


Figure 3: Combining execution and adaptation of TMK models; this is a high-level sketch of the process of SIRRINE.

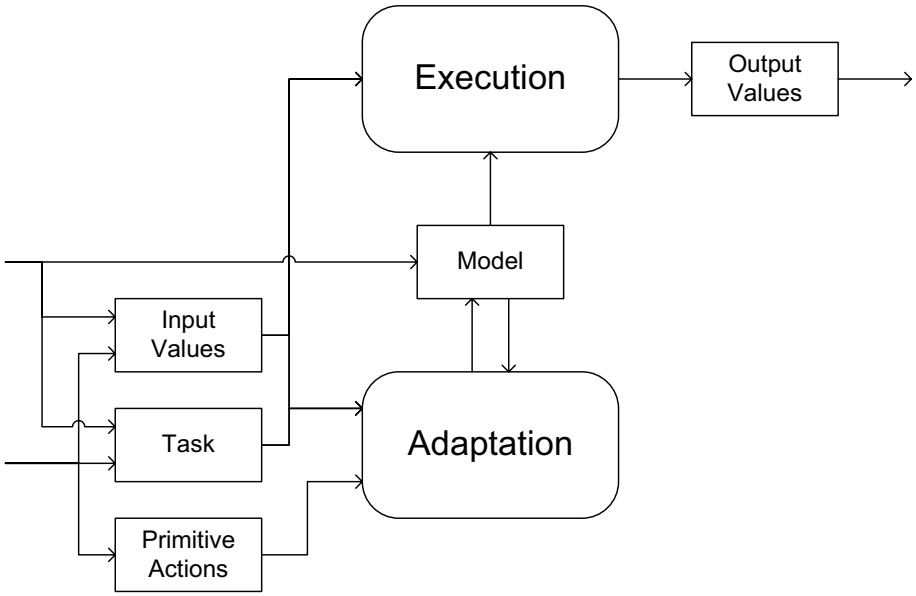


Figure 4: An alternative process; an unimplemented task requires adaptation before it can be executed.

In the second path, when an unimplemented task is requested, adaptation takes place first and then the adapted task is executed. In either case, execution can be followed by additional adaptation and execution if necessary. Note that the REM process is a generalization of the SIRRINE process presented in Figure 3.

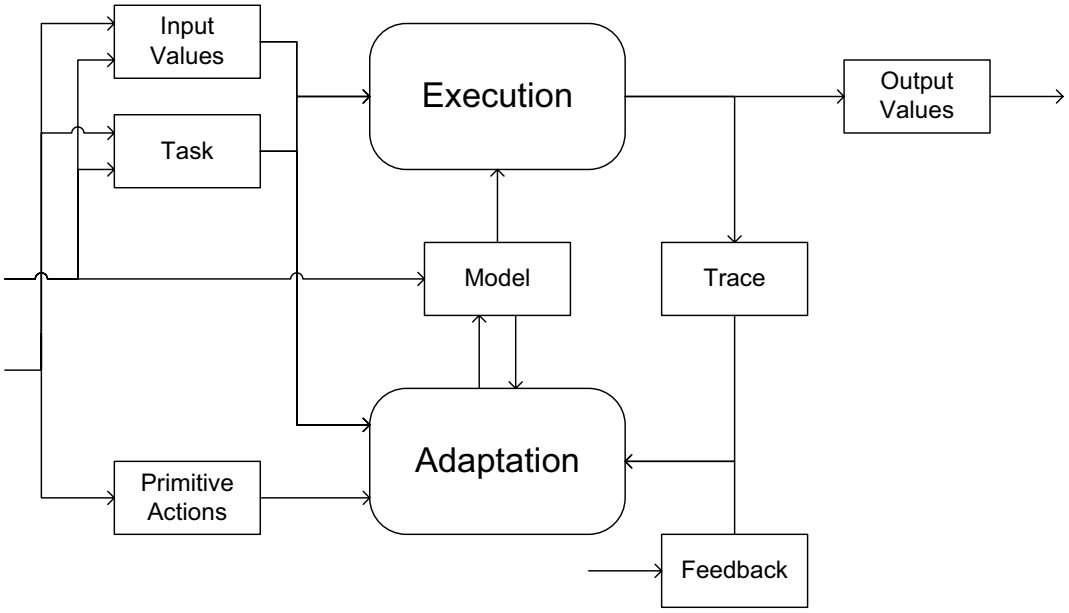


Figure 5: A high-level sketch of the processes of REM. This combines the approaches in Figures 3 and 4 into a single architecture.

1.3.1 Model Execution

Figure 6 shows a more detailed view of the execution portion of the reasoning architecture presented in the previous section. There are three inputs to the process. The first is a task to be performed. The second is a set of input values for that task. The third is a model of the agent; note that the model must contain at least one method for the task in order for execution to take place. The process begins with the execution of the task, which in turn leads to execution of a method, which then leads to additional execution of lower level tasks, etc. At the lowest point in the recursion, primitive tasks are encountered which can be directly executed. Once the main task (including its method, subtasks, etc.) has been executed, the process is done. The process returns whatever output values were produced during execution and a trace of the steps that were taken (i.e., what methods were selected, what subtasks were chosen for those methods, and what inputs and outputs went to and from those tasks).

Note that there are portions of both the task execution process and method execution process which involve selecting among alternatives. If a task has multiple methods, task execution must select among those methods. Similarly, if a method has multiple paths through it leading to different subtasks, method execution must select among those paths to find the next subtask to perform. It is generally expected that in most situations, the model will precisely specify exactly which option to choose when selecting methods or subtasks. TMKL includes slots for specifying applicability conditions in both methods and links to subtasks. Typically if a task has multiple methods available or a method has multiple transitions to the next subtasks in some state, then the applicability conditions will be mutually exclusive or at least rarely overlapping. This tendency is simply a reflection of the fact that a model of an agent is intended to specify how that agent works and if the model allows for multiple possible behaviors, then it is not completely specifying how the agent works. However, TMKL does *allow* the model to leave multiple alternatives open, i.e., to leave some of the description of the process unspecified. When REM encounters a situation in which it there is more than one option available, it selects one using Q-learning [Watkins and Dayan, 1992], a simple and popular reinforcement learning technique. Because Q-learning is only run in the generally infrequent event that multiple alternatives are available, it only needs to be used for a very small number of decisions within a given agent. By limiting the number of decisions made by Q-learning it is possible to avoid the efficiency concerns that can make this technique ill-suited to large, complex problems.

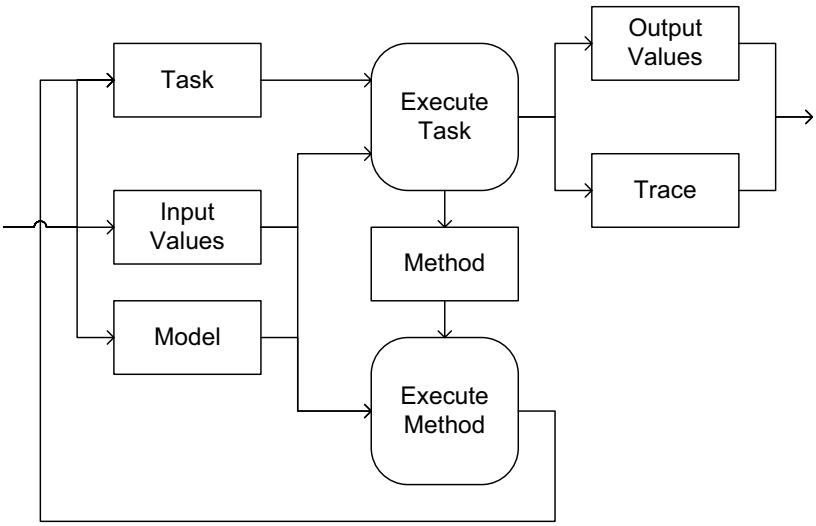


Figure 6: The REM model execution process.

1.3.2 Model Adaptation

In this work we have explored four general varieties of adaptation. See Chapter 6 for more information about each of these.

- **Failure-Driven Model Transfer** (Figure 7) Given a model, a trace of a failed execution of that model, and feedback (if any) which has been provided by the user, modify the process to instead succeed for the same input values. Within the area of failure-driven transfer, this work includes a generic processes for analyzing a trace in the context of user feedback and for selecting potential localizations for a failure. In addition, this work includes two repair mechanisms which are used to repair failures: fixed value production and generative planning repair. The latter uses the same process as the generative planning adaptation strategy (below) but does so in the context of a localized subtask rather than for the main task.

- **Proactive Model Transfer** (Figure 8): Given a new, unimplemented task to perform, retrieve a similar implemented task, and adapt the process for the similar task to instead address the new task. Within the area of proactive transfer, this work includes one mechanism: relation mapping. That mechanism is used to transfer methods from existing tasks to new tasks when the effects of the tasks are directly connected by some relation.

- **Situated Learning** (Figure 9): Given a new, unimplemented task to perform create a method in which all possible actions can be attempted at any time (thereby leaving all of the decisions regarding how to act open for the Q-learning portion of the execution process).

- **Generative Planning** (also Figure 9): Given a task to perform, use a traditional generative planning algorithm to construct a sequence of actions which accomplishes the end state of this task. Note that in addition to the use shown in the figure (i.e., adapting an entirely new task prior to execution), the generative planning mechanism may also be used to fix a particular subtask during failure-driven model transfer, as described above.

REM and SIRRINE are both able to perform adaptation of the first kind listed above (failure-driven transfer). In addition, REM is also able to address the other three types of adaptation. Figure 10 shows a more general architecture for adaptation combining all four of these approaches. This combination is embodied in the adaptation mechanism within REM.

1.4 Dissertation Outline

This dissertation is divided into three major portions. Chapter 1 discusses the essential ideas for this work. Chapters 2 – 6 provide the technical details of the knowledge structures and processing strategies developed in this research.

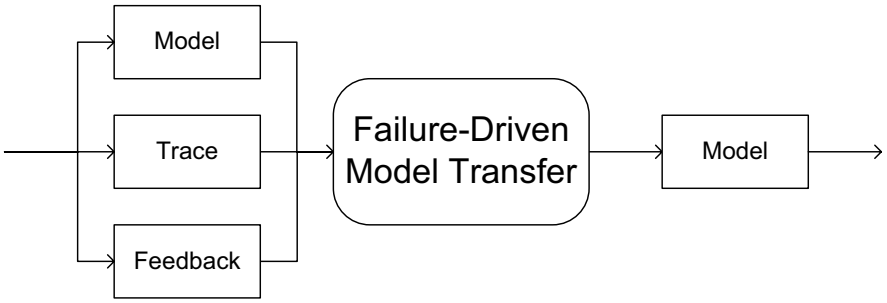


Figure 7: Adaptation of an implemented task, given the model of the task and the results of a specific execution.

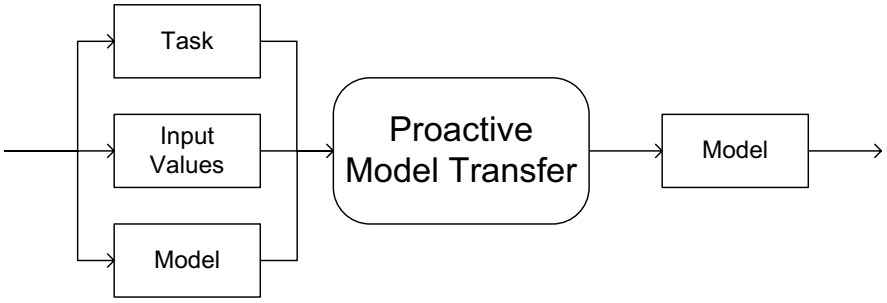


Figure 8: Adaptation of an unimplemented task, using a model of some existing, similar task.

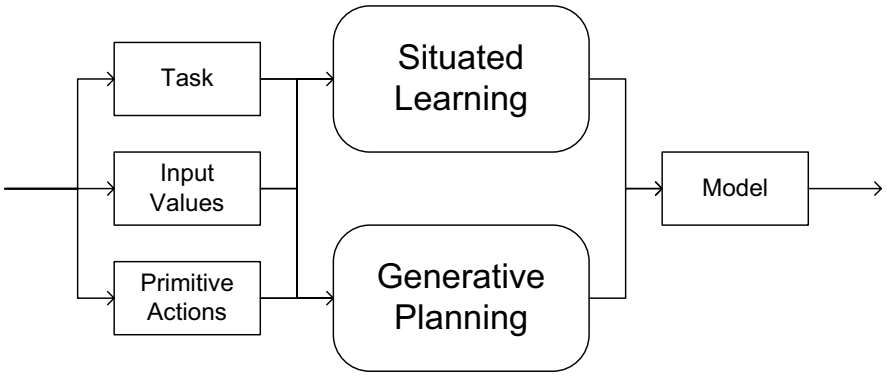


Figure 9: Adaptation of an unimplemented task without any relevant similar task available.

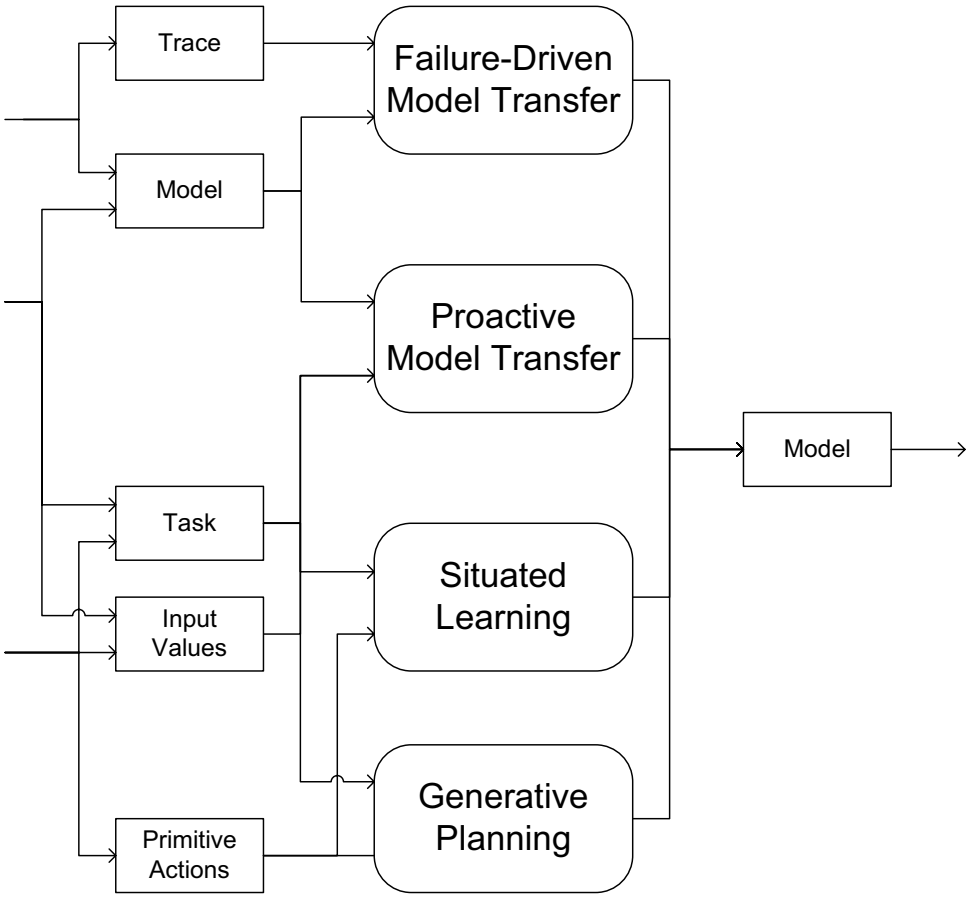


Figure 10: The overall REM adaptation process; this is a combination of Figures 7, 8, and 9.

Chapters 7 – 10 then present the consequences and significance of this work. More precisely, the contents of the chapters are:

1) Introduction: Discusses the challenges and provides a high level reasoning architecture.

2) Agent Models: Presents one of the two major aspects of the knowledge in this research, general models of the overall behavior of the agents.

3) Trace Information: Presents the other major aspect of the knowledge in this research, specific traces of particular executions of the agents.

4) Example Agents: Describes some agents which have been modeled and reasoned about.

5) Execution: Presents one of the two major aspects of the processing in this research, execution of agents that are encoded in the modeling language.

6) Adaptation: Presents the other major aspect of the processing in this research, adaptation of agents that are encoded in the modeling language.

7) Evaluation: Describes the experiments which have been run and provides some analysis of the complexity of the various mechanisms.

8) Related Research: Compares and contrasts the research with similar projects in Artificial Intelligence and Software Engineering.

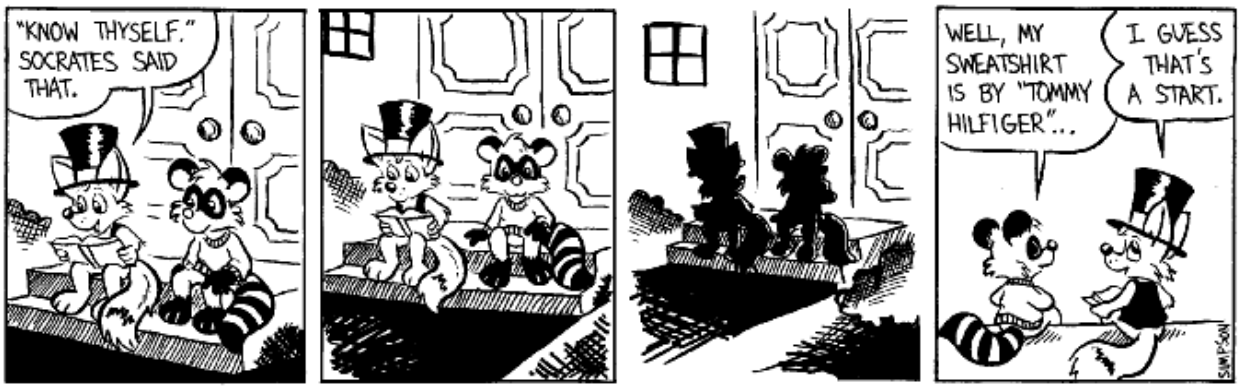
9) Future Work: Discusses current limitations and future enhancements which could be made to this work.

10) Conclusions: States the major claims and contributions.

Together these chapters provide a theory of models, reflection, and adaptation. This theory does build on existing work. The models presented in Chapter 2 fall within the existing TMK paradigm, but add a variety of new features not found in earlier work within this paradigm. Most notably, this new formalism provides particularly complex logical expressions in tasks and methods, which are a crucial element in the level of depth to which a model can represent what effects some portion of a process has. Traces, as described in Chapter 3, are also not a new idea; again, this research extends past work in this area. Among the example agents presented in Chapter 4 are a mixture of new and existing agents, demonstrating the effectiveness of this theory both in addressing new problems and in handling problems that are already known to be significant. The execution algorithms in Chapter 5 resemble a variety of existing execution mechanisms, but are updated to support the novel features in the new TMK formalism. Furthermore, the execution algorithms also encode a capability which has not been found in past work on model-based self-adaptation: reinforcement learning. The reinforcement learning technique used here (Q-learning) is a well-established one; however, the mechanism for integrating this technique with reasoning using symbolic self-models is new and provides an effective combination of the generality of reinforcement learning and the efficiency of knowledge-intensive reasoning. There is also considerable novelty in the techniques presented in Chapter 6 for adaptation of self-models. These techniques are: situated learning, generative planning, proactive model transfer by relation mapping, failure-driven model transfer by fixed value production, and failure-driven model transfer by generative planning. All of these mechanisms are new; they do, however, include some portions which are not new (especially the generative planning adaptation mechanism, which invokes an off-the-shelf planning system for part of its process). Lastly, the results and analysis in Chapters 7 – 10 clarify and validate the technical contributions presented in the earlier chapters. The primary claim of this dissertation is that the combination of these new and existing techniques for reflective model-based adaptation enables agents to reason flexibly with an effective balance between knowledge requirements and computational costs.

Chapter 2

Agent Models



The modeling approach used in this research is known as Task-Method-Knowledge or TMK. The basic idea behind TMK is that descriptions of what an agent does (tasks) and how it works (methods) are expressed in terms of the information it uses and processes (knowledge). Task contain links to methods which accomplish them and methods contain links to lower level tasks which are required to create the overall effect. Thus tasks and methods are arranged in a hierarchy: tasks refer to methods which accomplish them and methods refer to tasks which are a part of them, all the way down to primitive tasks which have a direct specification of their effects.

The details of TMK presented in this chapter are the ones in REM. The specific implementation of the TMK modeling approach encoded in SIRRINE is not presented in detail here; instead, see [Murdock and Goel, 1999b]. The variant of TMK used in REM is a refinement of the ones that are used in SIRRINE and earlier systems. REM's variant of TMK is called TMKL (TMK Language).

TMKL is built on top of Loom [Brill, 1993, MacGregor, 1999], a widely used knowledge representation system. Loom represents knowledge in the form of concepts, instances, and relations, as in a traditional frame system or semantic network. Loom also provides a variety of knowledge manipulation features including classification, querying, truth maintenance, etc. Some particularly fundamental operations that Loom provides include the ability to `tell` Loom that some assertion holds, to `ask` if some expression is true, and to `retrieve` some abstractly characterized value.

The elements of TMKL (such as tasks, methods, etc.) are represented as concepts within Loom. Furthermore, the knowledge items that the tasks and methods manipulate are also encoded in Loom. Thus knowledge about the domain and knowledge about the process are all represented in the same formalism and are accessed with the same mechanisms. TMKL provides a set of concepts and relations within Loom which constitute an ontology for representing knowledge and processing in an integrated manner. TMKL also provides a set of specialized syntactic forms for encoding models in a way that is easy to write and read.

There are two major uses of models in this research. Execution (described in Chapter 5) involves taking a model of an agent and a set of inputs and producing the effect indicated by the model. Adaptation (described in Chapter 6) involves taking a partial or complete model and producing a revised model which accomplishes a different effect. This chapter focuses on the language for representing the models. Examples of TMKL are used frequently in this chapter. Many of these examples are taken from a web browsing agent; the complete TMKL representation of this

agent is provided in the Appendix at the end of this dissertation.

2.1 Diagrams of TMKL Models

One way to convey an overview of the key elements of a model is to draw a diagram of that model. This section presents some ways to graphically depict a TMKL model. This presentation is intended to serve two purposes. The first purpose of this section is to provide an overview of the different aspects of these models before getting into the technical details of how the models are represented within REM. The second purpose of this section is to describe what information is included in these diagrams and what is left out; this description is particularly important because many of the references to models in later chapters primarily involve diagrams rather than formal TMKL code because diagrams often clearer and simpler.

Figure 11 presents a particularly detailed notation for TMKL models. Tasks and methods, on the right side of the figure, provide the description of what the agent does and how it does it. A line from a task down to a method indicates that the task can be accomplished by that method. Methods contain state transitions machines within them; each state in the machine directly identifies a lower level task which is to be performed in that state. The states and transitions encode how the computation works in terms of what effects need to be combined to form the overall effect (i.e., they compose the behaviors of the subtasks into the behavior of the main task). At the bottom of the hierarchy of tasks and methods are primitive tasks which are not further decomposed.

Concepts and relations, on the left side of the figure, provide the description of the knowledge that the agent uses and produces. A concept is an abstract description of a kind of knowledge and a relation is an abstract description of a kind of connection between instances of a concept. Parameters provide a bridge between the knowledge and the tasks and methods; a task has input parameters (depicted by arrows leading from parameters to tasks) and output parameters (depicted by arrows leading from tasks to parameters). The parameters are also linked to the concepts for which their values must be instances.

What does not appear, even in these extremely detailed diagrams, is the information inside the boxes. For example, tasks include descriptions of conditions which must be true before and after execution. The various sorts of information which is encoded within the different kinds of elements of a TMKL model are described in detail in the other sections of this chapter.

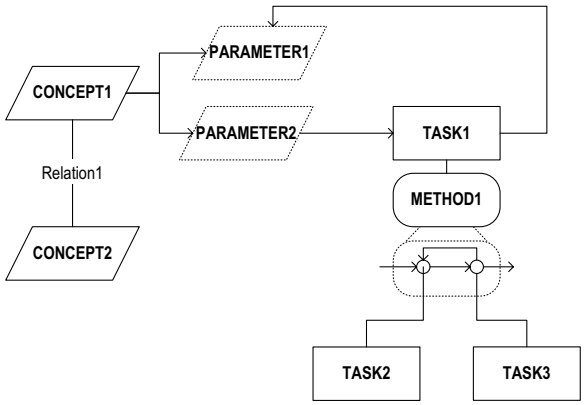


Figure 11: An abstract form of a very detailed diagram of TMKL model information. Presents information about tasks, methods, knowledge, and the links between them.

Unfortunately, the notation presented in Figure 11 tends to be overwhelmingly cumbersome for all but the simplest of TMKL models. Many tasks have several input and output parameters, and many parameters are inputs for some tasks and outputs to others. Consequently, this notation demands an enormous number of links between tasks and parameters. This problem is partially addressed in the notation used in Figure 12. In this figure, the parameters have been dropped. Instead, there are links which run directly between concepts and tasks, and it is left implicit that there are parameters in the model which instantiate those links. In addition, this figure omits the names of the relations, which can also add to the clutter of a diagram. Thus this notation includes a relatively broad overview of

the kinds of knowledge in the agent and what kinds of knowledge relate to other kinds of knowledge.[1]

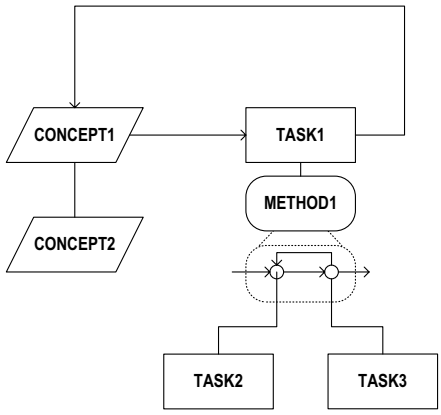


Figure 12: A slightly less detailed diagram of TMKL model information; parameters and the names of relations are omitted. This version still presents tasks, methods, and knowledge.

In practice, however, even the simplified notation in Figure 12 tends to be too detailed for practical use on the agents which are described in this dissertation. In particular, there are still too many links between concepts and tasks. Removing the parameters from these links increases the number which can be squeezed in to a diagram, but still leads to figures which are generally too cluttered to be easily described and understood. Consequently, most TMK diagrams that have appeared in this and other work have not attempted to explicitly depict the concepts and how they connect to the tasks and methods. Instead, the diagrams have focused on the tasks and methods themselves; information about the knowledge used by the tasks and methods have appeared only in the textual descriptions which accompany the figures.

Figure 13 presents a notation which excludes all concept and relation information and focuses on tasks and methods. This notation has been used in a variety of recent TMK-related papers such as [Murdock, 1998b, Murdock and Goel, 2001]. It seems to present a reasonably comprehensive overview of the tasks and methods in a model and does so in a way which precisely reflects the way that methods combine subtasks to form a behavior which supports an overall task. Note, however, that the diagrams of the state-transition machines in this notation do provide more details regarding the internal workings of the methods than may be necessary for some sorts of models. The overwhelming majority of methods which have been developed in this and other work involving TMK have involved simple sequences or loops. The power and flexibility of state-transition diagrams seems excessive for such simple methods. In particular, it is possible to depict sequences and loops among subtasks by simply drawing arrows between the subtasks themselves, without explicitly representing the states which connect the subtasks to the methods. Figure 14 presents such a notation; arrows are drawn from methods down to the first subtask of that method and then arrows among subtasks indicate the ordering of those tasks within the method.

The most substantial drawback to the notation presented in Figure 14 is the fact that it doesn't handle complex relationships between tasks and methods very well. In contrast, the more complex notation in Figure 13 precisely mirrors the internal representation within TMKL of reasoning states and transitions (described later in this chapter) so whatever kinds of relationships can exist among these elements in the model can be directly represented in the diagrams. For example, if two different methods both use the same subtasks in different orders, then this is easily handled using the Figure 13 notation because the states and transitions which control the ordering of the subtasks are depicted separately from the subtasks themselves. In contrast, however, any subtasks which occur in more than one method in the Figure 14 notation must be represented separately for each method in which they occur. This contrast is illustrated in Figures 15 and 16, which use the two notations to present an example agent.[2] Both diagrams present two methods for the Move Tower task. These methods involve the same two subtasks; the difference between the methods is that one invokes each subtask only once while the other invokes them repeatedly in a loop. Figure

[1]Note that the issue of how to represent concepts and relations graphically has been addressed many times in the past. For example, the CommonKADS [Schreiber et al., 2000] methodology uses UML class diagrams for this purpose, treating concepts as classes and relations as links between classes. This seems to work fairly well; the issue is not addressed in more detail because the research presented in this dissertation has focused more on representing and reasoning about the process portions of the model (tasks and methods) than on issues relating to concepts and relations.

[2]This example agent is described in detail in Section 4.3. The discussion here focuses only on the diagram, not the agent itself.

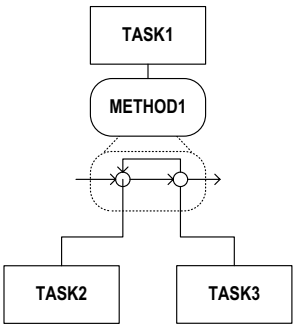


Figure 13: An even less detailed diagram of TMKL model information; knowledge is omitted and only tasks and methods are presented.

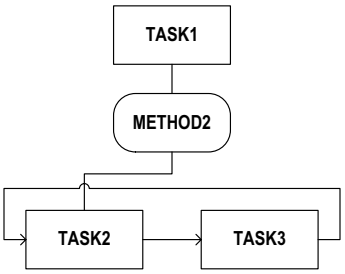


Figure 14: A particularly simple diagram of TMKL model information. Knowledge is omitted, and the details of the methods are simplified.

15 uses the state-transition diagrams to make this fact directly apparent. Figure 16, on the other hand, presents the same two tasks twice. Anyone viewing the latter diagram must determine that the subtasks match because they have the same names; this is not hard to do for this relatively simple figure, but a very complex diagram in which many tasks appeared in more than one place could be very confusing.

In practice, the models which have been developed in this research have very few tasks which are used in more than one method. Furthermore, some of these models are large enough that trying to include state-transition diagrams would create an excessively cumbersome figure. Consequently, the simplest of the notations presented in this section (the one from Figure 14) is used in most places in this dissertation.

2.2 Tasks

A description of a task encodes what a piece of computation is intended to do. Table 1 presents the schema for a task in TMKL, i.e., a list of the slots in a TMKL representation of a task and the kinds of information which goes into those slots. Note that each of these slots corresponds to a relation encoded in Loom which maps from tasks to values of the type for that slot. TMKL uses a convention in which a relation for slot x in concept y is given the name y>x. For example, there is a relation task>input which maps a task to zero or more parameters.

The input and output slots of a task identify the kinds of knowledge the task uses and produces. Both input and output take the form of zero or more parameters. Note that the parameters are defined separately (in Section 2.4) and refer to the abstract concepts which they instantiate.

The given and makes slots contain logical expressions which are required to be true before and after the task is performed. These expressions are written using Loom's formalism for queries (specifically, they are nested lists of symbols of the form used by Loom's `ask` operation). This formalism combines concepts, relations, and values using both basic logical operators and more advanced features such as universal and existential quantification, numerical comparison, and even invocation of arbitrary LISP code. The given and makes slots in a TMKL model serve two major purposes. First, during execution they allow REM to detect when a task cannot be attempted or has been attempted unsuccessfully (by checking whether the given and makes expressions are true before and after executing the task). Second, the given and makes conditions describe what effect the task is intended to have; during adaptation

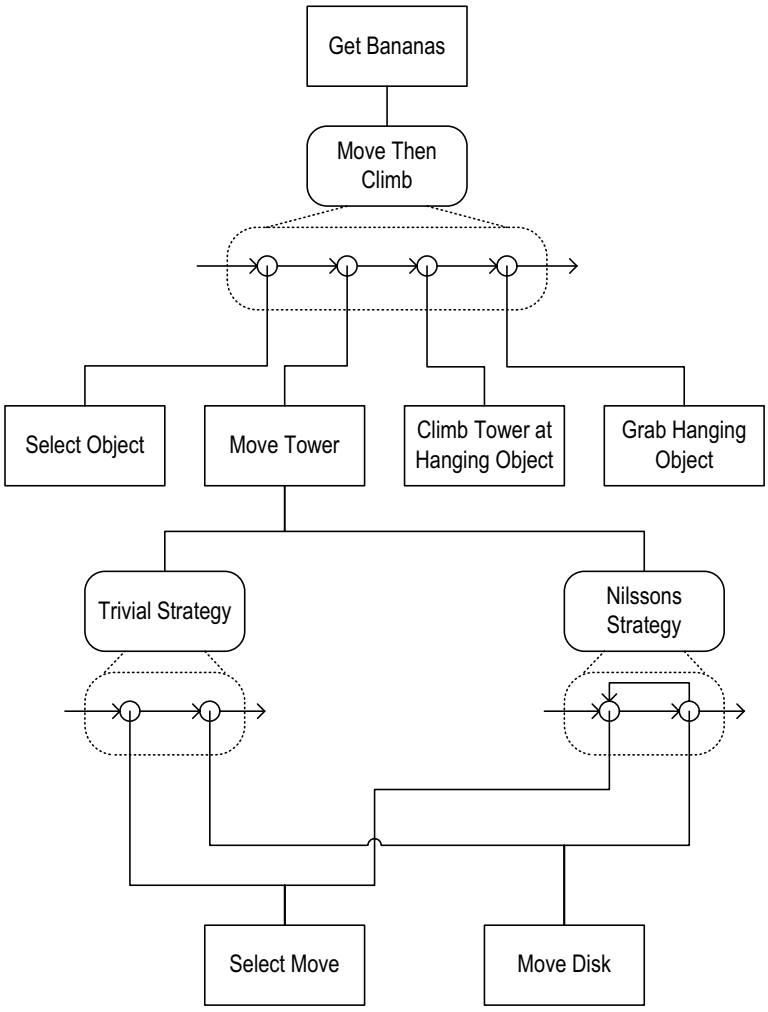


Figure 15: A diagram of a monkey / banana / Tower of Hanoi agent. This diagram presents the state-transition machine for the methods, as in Figure 13.

Concept **Task**		
Slot	*Type*	*Quantity*
input	parameter	0+
output	parameter	0+
given	logical expression	0-1
makes	logical expression	0-1
by-mmethod	method	0+
by-procedure	procedure	0-1
asserts	logical assertion	0-1
binds	parameter value binding	0+

Table 1: Schema for tasks in TMKL

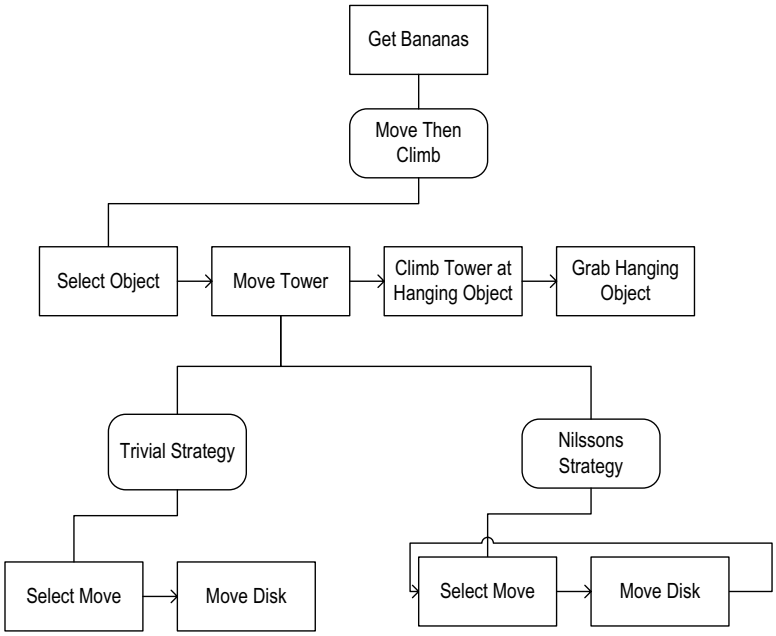


Figure 16: This is the same agent which appears in Figure 15 but with the state-transition machine omitted, as in Figure 14.

this information is useful in deciding whether that effect needs to be used and/or modified in the adapted agent. These two applications of this information are explored in more detail in Chapters 5 and 6 respectively.

In addition, some tasks contain information about how they are implemented. TMKL allows three types of tasks, categorized by the information that they contain about implementation:

- **Non-primitive tasks** have a slot, by-mmethod,[3] which contains any number of methods which can accomplish that task.
- **Primitive tasks** have some direct representation of the effects of the task. A primitive task in TMKL must have a value in at least one of these three slots: by-procedure, asserts, and binds. The by-procedure slot contains a reference to a LISP procedure which accomplishes the desired effect; note that this LISP procedure can, in turn, invoke code in other programming languages such as C using ordinary external function calls. The asserts slot contains a logical assertion, i.e., a nested lists of symbols of the form used by Loom's `tell` operation. When a task with the asserts slot filled is invoked, the assertion in that slot is made to be true. Lastly, the binds slot contains any number of connection between a particular parameter (which should appear in the output slot for that task) and a Loom specification of a value (in the form of arguments to the Loom `retrieve` operation). When a primitive task with a binds slot filled is involved, the mentioned parameters are made to have the values defined by the associated expression. Note that while the by-procedure slot is powerful enough to encompass the other two ways of implementing primitives (since making assertions and binding values can be done in LISP), the other two slots are generally more useful for adaptation. An adaptation technique can directly and easily reason about and modify logical assertions and parameter bindings; however, accessing and modifying the internals of an arbitrary LISP procedure is typically not feasible (and REM does not attempt to do so).
- **Unimplemented tasks** have no information about how they are to be accomplished, either in the form of methods or in the form of primitive information. Such tasks cannot be immediately executed; they must,

[3]Note that the misspelling of the word "method" as "mmethod" is deliberate here and is used consistently throughout TMKL. It is intended to avoid naming conflicts between functions and values which involve TMKL methods and those built in to Common LISP which involve CLOS methods. This convention was derived from Autognostic [Stroulia and Goel, 1996, Stroulia and Goel, 1997] where it was presumably motivated by the same issues. One obvious alternative to this misspelling would be to simply allow names to conflict and use LISP's package mechanisms to resolve these conflicts. This would be mildly inconvenient but would presumably not create any other problems. The convention used in this thesis is to use the term "method" for all descriptions of methods in both the text and in pseudo-code algorithms *except* when explicitly referring to a specific slot, such as by-mmethod.

```
(define-task communicate-with-www-server
  :input (input-url)
  :output (server-reply)
  :makes (:and
           (document-at-location (value server-reply) (value input-url))
           (document-at-location (value server-reply) local-host))
  :by-mmethod (communicate-with-server-method))
```

Table 2: An example of a task in TMKL, defined using a specialized TMKL form.

instead, be adapted into either non-primitive tasks or primitive tasks. An unimplemented task has no value in any of the by-mmethod, by-procedure, asserts, and binds slots.

Table 2 presents an example TMKL task called communicate-with-www-server. This task is the one in the web browser which retrieves a document from a server. The task has one input, a URL for the document, and one output, the information retrieved from the server. There is a makes expression which requires that the reply is located at both the location specified by the URL and (now) at the local host. This task has only one method, which decomposes the task of communicating with the server into lower level subtasks.

The makes expression in communicate-with-www-server is consistent not only with getting a document from the web but also with creating a document and putting it on to a web; thus the output requirement for this task is underconstrained. It is very common for given and makes expressions to be underspecified in TMKL tasks, especially high-level tasks. In fact, it would often be overwhelmingly difficult to completely specify all of the effects of a high-level task (because such a task can involve many subtasks with alternatives and loops among them). Underspecified given and makes expressions do not cause a problem during routine operation of an agent because the agent does not need to figure out how to address those expressions; the methods (or primitive implementation) of the task specify the process. However, underspecified given and makes expressions can be a problem during adaptation. For example, if the generative planning repair strategy presented in Section 6.4.1 were used to build a method for this task, it could produce one which did not accomplish the actual desired effect. This issue illustrates the graceful degradation of REM under varying knowledge conditions: REM can perform routine execution of agents with limited descriptions of the tasks, but to the extent that those descriptions are limited, REM's adaptation may be less effective.

Table 3 shows the same task represented directly using the underlying Loom mechanism for encoding knowledge items. This table is presented here to provide an example of how the specialized forms for TMKL (such as define-task from Table 2) are translated into Loom. The details of the Loom syntax are not crucial here; see [Brill, 1993] for more information on that topic. What is important to realize is that encoding a description of a task in a specialized TMKL form creates the same effect as encoding the same information using a generic formalism for representing assertions. This is a key strength of TMKL: it provides both the convenience of specialized forms and the power of general purpose assertions. It is expected that most of the information in a TMKL model will be encoded using the specialized forms; however, if the author of a model has some information which does not fit into those forms, that information can be encoded as arbitrary assertions. Furthermore, REM can reason about information encoded in both of these ways interchangeably. For example, when REM needs to know what parameters fill the output slot of a task, it can simply issue a query using Loom which requests that information, and it will be found regardless of whether it was entered using the :input line of the specialized TMKL define-task form or a generic :filled-by assertion over the task>input relation. In either case, the internal representation is the same.

2.3 Methods

Methods are behavioral elements: a description of a method encodes how a piece of computation works. Table 4 shows the schema for a method in TMKL. A TMKL method is described by a provided slot which specifies facts which must be true for the method to operate and an additional-results slot which specifies consequences of the operation of the method. Note that these slots are not intended to specify the overall function of the method (that information is encoded in the description of the task which the method addresses); rather these slots are intended to specify

```
(tell
  (:create communicate-with-www-server task)
  (:about communicate-with-www-server
    (:filled-by task>input input-url)
    (:filled-by task>output server-reply)
    (task>makes
      '(:and
         (document-at-location (value server-reply) (value input-url))
         (document-at-location (value server-reply) local-host))))
    (:filled-by task>by-mmethod communicate-with-server-method)))
```

Table 3: An example of a TMKL task represented directly in Loom without the use of the specialized define-task form.

Concept **Method**		
Slot	*Type*	*Quantity*
provided	logical expression	0-1
additional-result	logical expression	0-1
start	transition	1

Table 4: Schema for methods in TMKL

incidental requirements and results which are specific to a particular way of accomplishing the overall effect. For example the task of robot navigation might have a given state in which a robot is in one location and a makes state in which the robot is in another. One method for this task, case-based navigation, might a provided condition that a similar case is in memory, and an additional-result condition that a new case has also been stored in memory. Other methods might have provided clauses which involve availability or a map or of environmental sensors, etc. Note that the provided slot is particularly essential for execution: the information in that slot is used to determine what methods for a particular task are applicable in any situation. Furthermore, both the provided and additional-results slots are potentially useful during adaptation; specific information about the requirements and effects of the method can be relevant to analyzing and modifying that method and those around it.

In addition to the provided and additional-results slots, a method description also contains a slot called start which contains the first transition in a state-transition machine that describes the operation of the method. Table 5 shows the schemas for transitions in this machine. Each transition has a provided clause, similar to the one found in a method; if a state has multiple outgoing transitions, some may only be applicable in some situations. In addition, each transition links any number of pairs of parameters; for example, if a task which finds a document has an output called found-document and a task which displays that document has an input called document-to-display then a method which finds and then displays a document would have a transition which links these two parameters. Lastly a transition can have a next reasoning state which indicates what happens after the links are made; if the transition has no next value, then it is a terminal transition, i.e., when it is reached, the method is finished.

Table 6 shows the schema for reasoning states. A reasoning state has exactly one subtask which is executed in that state and at least one transition to follow. When a particular reasoning state is encountered during the execution of a method, it's subtask is executed (which may involve the execution of a lower level method, etc.) and then one of the transitions whose provided condition is met is selected. For more details on this process, see Chapter 5.

Note that a reasoning state is distinct from both a knowledge state and a world state. A reasoning state is a unit processing within a method. A knowledge state is a set of information which occurs at some point in time. A world state is a set of external circumstances which occur at some point in time. The three kinds of states are certainly related: for example, when a particular reasoning state is reached in an agent, that agent will have a certain knowledge state, and a certain world state will exist outside the agent. Knowledge states are not represented within a TMKL model because they are specific to a particular execution; REM's representation of knowledge states appears

Concept **Transition**		
Slot	*Type*	*Quantity*
provided	logical expression	0-1
next	reasoning state	0-1
links	parameter binding	0+

Table 5: Schema for transitions in TMKL

Concept **Reasoning state**		
Slot	*Type*	*Quantity*
subtask	task	1
transition	transition	1+

Table 6: Schema for reasoning states in TMKL

in Chapter 3. REM does not directly represent or reason about world states at all because the emphasis of REM is on reflection, i.e., reasoning about the agent rather than about the environment. However, virtually all agents do interact with some sort of external environment and most have some kinds of knowledge about that environment. Thus some aspects of the world state are likely to be reflected in an agent's knowledge state, which REM does represent and reason about. REM doesn't reason about the world but it does reason about the agent's reasoning about the world.

Table 7 shows a relatively simple method, encoded in a specialized TMKL form. This method is the one in the web browsing agent which displays a document using an external viewer. It has a provided condition which states that the value of the tag retrieved from the server must not be in the fixed set of tags for documents which can be displayed internally. In addition, the method has a series term which contains a list of subtasks which are invoked in the specified order. Internally, this term is translated into a combination of transitions and reasoning states which accomplish that behavior. In particular, this translation provides transitions named external-display-t1 through external-display-t4 and states named external-display-s1 through external-display-s3. Each of the reasoning states has the listed subtask. The start slot for the method contains external-display-t1. Each of first three transitions each points to the equivalently numbered state and the fourth transition has no next state. There are no provided conditions or links between parameters.

A more complicated method, make-plan-node-children-method, is presented in Table 8. This method is the one from the ADDAM disassembly planning agent[4] which constructs the children of a new node in a hierarchical plan based on the children of the related node in the existing hierarchical plan. This method operates in a loop; it steps through the list of children of the current node in the existing plan developing a set of equivalent new nodes for each of those children. Furthermore, the method is recursive: it is a method for a task called make-plan-node-children which is a subtask of the method for the add-plan-mappings which is, in turn, a subtask of make-plan-node-children-method.

[4]See Section 4.1 for more details regarding ADDAM.

```
(define-mmethod external-display
  :provided (:not (internal-display-tag (value server-tag)))
  :series (select-display-command
           compile-display-command
           execute-display-command))
```

Table 7: A relatively simple method represented in TMKL.

```
(define-mmethod make-plan-node-children-mmethod
  :series (select-child-plan-node
             make-subplan-hierarchy
             add-plan-mappings
             set-plan-node-children))
(tell (transition>links make-plan-node-children-mmethod-t3
                        equivalent-plan-nodes
                        child-equivalent-plan-nodes)
      (transition>next make-plan-node-children-mmethod-t5
                       make-plan-node-children-mmethod-s1)
      (:create make-plan-node-children-terminate transition)
      (reasoning-state>transition make-plan-node-children-mmethod-s1
                                  make-plan-node-children-terminate)
      (:about make-plan-node-children-terminate
              (transition>provided
               '(terminal-addam-value (value child-plan-node)))))
```

Table 8: A more complex method which cannot be entirely represented within a specialized TMKL form. Additional Loom assertions allow more elaborate control.

The representation of this method in TMKL begins with a specialized form in which the subtasks of the method are arranged in series. However, the details of the method are too complex to encode in the define-mmethod specialized form and thus must be stated directly as Loom assertions. The five assertions in the `tell` operation have the following effects, respectively:

- The transition which leads from the state with the make-subplan-hierarchy subtask to the state with the add-plan-mappings subtask is given a links slot. This slot connects the equivalent-plan-nodes parameter which is an output of the child-equivalent-plan-nodes parameter which is an input to a later task.
- The transition leading out of the state with the set-plan-node-children subtask is made to connect back to the state with the select-child-plan-node subtask. This creates a loop in the process.
- A new transition called make-plan-node-children-terminate is created.
- The new transition is asserted to lead out of the state in which select-child-plan-node is executed. Note that there was already a transition leading out of this state so there are now two possible alternatives after this state is completed.
- Lastly, the new transition has a provided condition which requires that the child-plan-node produced by select-child-plan-node represent the end of the sequence.

Since the make-plan-node-children-terminate transition is not given a next value, when that transition is selected, the method is done. Thus the overall behavior of this method is that it steps through each of the subtasks in order and then returns to the first subtask, repeatedly, until the first subtask returns a terminal value. The combination of the TMKL specialized form and the additional Loom information defines this behavior.

2.4 Knowledge

Knowledge is the foundation on which tasks and methods are built: a description of a task or a method is inherently interconnected with the description of the knowledge that it manipulates. The "knowledge" portion of a TMKL model consists of explicit representation of the concepts and relations used by an agent. Recall that REM uses Loom for its knowledge representation; thus Loom, itself, constitutes most of the knowledge aspects of TMKL. However,

```
(defconcept location)
(defconcept computer
  :is-primitive location)
(defconcept URL
  :is-primitive location
  :roles (text))
(defrelation text
  :range string
  :characteristics :single-valued)
(defrelation document-at-location
  :domain document
  :range location)
```

Table 9: Some sample concepts and relations from the web browsing domain.

TMKL does provide some additional constructs for representing kinds of knowledge which are particularly relevant to adaptive reasoning.

This section is broken into three subsections. The first briefly discusses the encoding of domain knowledge in Loom. The second describes some additional concepts and relations provided by TMKL on top of the standard Loom ontology. Finally, the last subsection describes parameters, which act as the bridge between the knowledge in an agent and its tasks and methods.

2.4.1 Domain Knowledge

Representation of domain knowledge in TMKL knowledge is based on Loom. Loom provides a syntax and ontology for defining elements of knowledge such as concepts and relations. Table 9 shows some example concepts and relations encoded in the Loom formalism; these concepts and relations serve as part of the knowledge portion of the TMKL model of the web browser. The first concept defined is location, i.e., a place within web space where a document may be. There are two subconcepts of location: computer and URL. The URL concept has a role associated with it: text. A role in Loom is implemented as a relation. The text role relates the URL to a string; it is constrained to have only one value (i.e., any given URL can have only one text). Lastly, there is another relation, document-at-location, which indicates that a particular document is at a particular location.

The example knowledge items illustrated in the table demonstrate many of the features of Loom which are useful for modeling domain knowledge for TMKL models. The definitions of location, computer, and URL demonstrate the nature of concepts and subconcepts in Loom. Loom's built-in ontology provides terms which not only create concepts and relations but also properties of them (e.g., single-valued) and relationships between them (e.g., the domain of a relation) as well as primitive data (e.g., the string in the text role of URL). This ontology provides a broad foundation for knowledge which can be useful across a wide variety of domains for a wide variety of purposes.

Loom provides representation and access of both knowledge and meta-knowledge. An example of a knowledge query is: (retrieve ?x (document ?x)); this query retrieves all instances of the document concept that are known in the current context. A similar meta-knowledge query is: (retrieve ?x (single-valued ?x)); this query retrieves all single-valued relations in the current context (including, for example, the text relation in Table 9). Thus Loom treats knowledge and meta-knowledge as fundamentally the same. This trait is particularly valuable for REM, because REM needs to explicitly reason about the kinds of knowledge that an agent has in order to adapt that agent. The relation-mapping algorithm in Section 6.3 provides a particularly dramatic example of reasoning of this sort.

Of course, a complete description of all of the capabilities of Loom is outside of the scope of this dissertation; see [Brill, 1993] and related documents for more on that subject. The examples presented in Table 9 are simply meant to illustrate a few of the kinds of things which can appear in the domain knowledge of a TMKL model.

Name	*Arity*	*Characteristics*
external-state-relation	1	
internal-state-relation	1	
external-definition-relation	1	
internal-definition-relation	1	
similar-to	2	symmetric
inverse-of	2	symmetric

Table 10: The additional ontological extensions that TMKL makes to Loom.

2.4.2 Additional Ontological Extensions

Loom's built-in ontology provides a broad and flexible basis representing both knowledge and meta-knowledge. It is certainly possible to represent a particular domain using only this ontology as a foundation. However, during the course of the development of REM, it became apparent that there are certain kinds of information which exist across domains and are particularly useful for the sorts of adaptive reasoning which REM does. These additional sorts of information have been added to REM as extra terms in the TMKL ontology.

Table 10 presents these additional terms. The table has three columns: one which states the name of the term, one which states the arity (i.e., the number of arguments to the term), and one which provides any additional characteristics of the term. The first four terms are unary and represent traits that a relation can have. An external state relation is one which exists in the environment outside the agent and represents a potentially changeable state. For example, in the domain of manufacturing, the position of a component is an external state relation: a position can be observed and changed in the environment. An internal state relation is one which exists within the agent and can be meaningfully changed. For example, the type of an action in a plan is an internal state relation: it is possible to change the plan directly within the agent. Definition relations involve information which is fundamental to the concept and thus cannot change; like state relations, definition relations can be either internal or external. For example, the text relation for the URL concept in the previous section is an internal definition relation; if the text of a URL were to change then, by definition, it would be a different URL.

The distinction between internal and external state relations is particularly important for adapting an agent because it determines what sort of effects an agent can produce simply by modifying its knowledge and what effects require action in the world. For example, if an agent first builds a plan with various types of actions (internal state) and then executes that plan to affect the position of objects (external state), it is possible to adapt that agent by directly modifying the actions in the plan thus indirectly affecting the positions of the objects. One can't directly change the agent's knowledge of positions to accomplish the desired effect; this would simply make the agent believe that the object was in a different position. However, changing the knowledge of the plan *is* changing the plan, because the plan only exists within the agent. Consequently, if the designer of an agent declares certain relations to be internal state relations or external state relations, REM can use this information to guide what sorts of changes can be made to the tasks and methods which affect those relations. More specifically, REM identifies a specific kind of task called a primitive action; a primitive action is a primitive task which has some postcondition (either a makes expression or a asserts assertion) which does not directly refer to an internal state relation. In other words, primitive actions are primitive tasks which affect the outside world, not the knowledge of the agent. Some of the adaptation strategies in Chapter 6 make explicit use of the set of primitive actions in an agent. Note that a declaration of internal or external for a relation is optional. REM can still execute an agent if some or all of the relations do not have these traits declared, but it may miss some opportunities for adaptation.

The remaining terms in Table 10 are similar-to and inverse-of. These terms denote binary (arity 2) relations which are symmetric (e.g., if X is known to be similar-to Y then Y is automatically known to be similar to X). The similar-to relation indicates that two pieces of knowledge have similar content and/or purpose; for example, two tasks which accomplish similar effects would be considered similar-to each other. The inverse-of relation indicates that two pieces of knowledge have content and/or purpose which are the negation of each other; for example, two tasks can be run sequentially to produce the same effect as running neither of them are inverse-of each other. These two terms are particularly valuable for adaptive reasoning process because they denote relationships which are specifically relevant to determining whether and how one computational element can be altered with respect to another computational element. For example, if similar-to holds between two tasks, then it may be useful to try to adapt a method for one

Concept **Parameter**		
Slot	*Type*	*Quantity*
concept	Loom concept	1
value	Loom instance	0-1

Table 11: Schema for parameters in TMKL

of the tasks to accomplish the other one, as needed.

Both similar-to and inverse-of are relations which can either be computed by the system or asserted directly by the developer of the agent. REM allows an agent to use information about similarity and inversion from either or both of these sources. In particular, Loom has a mechanism called implies which makes it possible to establish a condition under which some relation is automatically determined to hold. REM does not have any built-in implies rules for inverse-of but it does have a couple of these rules for similar-to. Furthermore, it is possible for a particular agent to contain additional implies rules for these (or any other) relations within a particular domain. This is a very valuable feature because some kinds of similarity or inversion may only be meaningful within a given domain. Thus, for example, if REM has an unimplemented task to perform and it is trying to find a similar implemented task, it can simply issue a query over the similar-to relation and this query seamlessly integrates knowledge from three sources: (i) direct assertions of similarity, (ii) similarity computed using REM's general purpose rules, and (iii) similarity computed using domain-specific rules.

While the ontology extensions presented in this chapter provide some important terminology for adaptive reasoning, it is important to note that the Loom built-in ontology also contains many key concepts and relations which also affect how adaptation can occur. Furthermore, any given domain is likely to contain additional concepts and relations which may be relevant to determining how an agent in that domain can be adapted. Because Loom provides a uniform framework for representing concepts and relations, REM is able to use terms built into Loom, additional terms provided by REM, and terms which are specific to a particular domain interchangeably. For example, one step of the Relation Mapping algorithm in Section 6.3 finds a relation which maps the effects of two different tasks; such a relation may come from Loom, REM, or a particular domain, and the algorithm is not affected by this distinction.

2.4.3 Parameters

Parameters act as the connection in TMKL from the tasks and methods to the knowledge. Recall from Section 2.2 that the input and output slots of a task contain a list of parameters. Furthermore, the given and makes conditions of a task and the provided and additional-results conditions of a method often refer to the values of parameters (for example, see Table 2 and Table 7). The definitions of the tasks and methods are inextricably tied to the knowledge that those tasks and methods process, and parameters form the bridge between these portions of TMKL.

Table 11 shows the schema for parameters. Each parameter has exactly one concept; furthermore, parameters which are bound have exactly one value (unbound parameters have no value). It is required that if a parameter has a value that this value be an instance of the parameter's concept. Note that the slot value is one which is accessed very often in the logical expressions within a TMKL model. As a convenience, the relation for that slot, parameter>value, has been given an alternative name: value. For example, the references to value in the given slot of Table 2 are actually referring to the parameter>value relation; the two forms can be used interchangeably.

Table 12 shows an example of a parameter being defined and given a value. The definition appears as a specialized TMKL form. The creation of the value and the binding of that value to the parameter is done using ordinary Loom assertions. The parameter defined here is input-url; this is the input parameter to the task presented in Table 2. The concept slot for this parameter is declared to be URL; the URL concept is defined in the domain knowledge portion of the TMKL model. The value that it is given is first created and the information in that value is filled in (in this case, that information is simply a text string). Lastly, the parameter>value relation is asserted to hold between the new value and the parameter.

```
(define-parameter input-url URL)

(tell
  (:create demo1-url URL)
  (text demo1-url "http://thesis.murdocks.org/thesis.pdf")
  (parameter>value input-url demo1-url))
```

Table 12: An example of the definition of a parameter along with the creation and assignment of a value for that parameter.

2.5 TMKL Summary

TMKL is a complex modeling language with many elements to it. Some key points to remember regarding the nature of TMKL are:

- The syntax of TMKL includes the syntax of Loom plus some specialized TMKL forms.
- The ontology of TMKL is divided into 3 parts: "T", "M", and "K".
  - The "T" portion of the ontology includes the concept **task** and the relations which form the slots for that concept (e.g., task>input).
  - The "M" portion of the ontology includes the concepts **method**, **reasoning state**, and **transition** and the relations which form the slots for those concepts.
  - The "K" portion of the ontology includes the default ontology for Loom, plus a few more additional terms for describing domain knowledge, plus the concept **parameter** (which provides a bridge to the "T" and "M" portions) and the relations which form the slots for that concept.

The representation of tasks, methods, and knowledge together form a model of an agent. Models of this sort, along with some additional knowledge, described in Chapter 3, provide a content account for the theory of reflective reasoning presented in this dissertation. This content provides the foundation for the execution and adaptation mechanisms described in later chapters.

Chapter 3

Trace Information

"Ozy & Millie" © 1999 D. C. Simpson

Models are the most substantial aspect of the knowledge that is used in the reflective reasoning processes described in this dissertation. A model of an agent describes that agent abstractly, in a way that is independent of any particular execution of that agent. However, it is also important to be able to reason about concrete examples of the operation of the agent; this requires knowledge about individual instances of execution.

Recall from Chapter 1 that the input to the agent architecture is a model and a knowledge state, and that the output is a possibly revised model, a new knowledge state, and a trace of execution. Traces and knowledge states are thus essential elements of this architecture. Unlike models, traces and knowledge states involve concrete examples of execution. The language for representing these items is fundamentally intertwined with the modeling language. In REM, the syntactic forms for building traces and knowledge states are consistent with the TMKL language for models; the information within these items can refer to each other. For example, a step in a trace refers to the piece of the model which was executed in that step.

Figure 17 is a diagram an example trace; the example involves the execution of the monkey and tower agent (described in Section 4.3) with a two disk tower. Contrast this diagram with the model diagrams which appear in Section 2.1. The trace only references those parts of the agent which were actually executed in the particular session which the trace records. For example, in the model the Move Tower task contains a method called Trivial Strategy. Because that method was not invoked during this execution, it does not appear in the trace. In addition, the trace contains no branches or loops; it simply describes the actual sequence of tasks and methods which occured. Lastly, the trace contains knowledge states which record the knowledge in the agent at each step of the process.

This chapter describes the elements of a trace and the additional knowledge referred to in traces. The first section describes knowledge states. The second section describes the various pieces of a trace. The final section discusses some special annotations which can be attached to a trace, denoting failures detected by the system or feedback provided by the user.

3.1 Knowledge States

A knowledge state is an explicit representation of the knowledge that an agent contains at a particular point during the execution of that agent. There are three main reasons why it is useful for REM to treat knowledge states as first

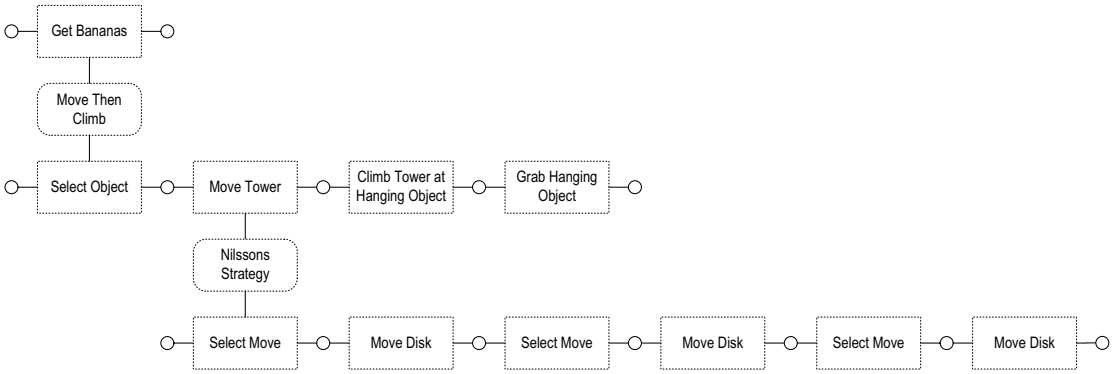


Figure 17: An example of a trace. The dotted rectangular boxes represent task traces and the dotted rounded boxes represent method traces. The small circles between task traces represent knowledge states. Omitted are reasoning state traces, transition traces, and trace annotations.

class objects:

1. REM contains a TMKL model of itself. This model can only describe how REM affects knowledge states if it has an explicit representation of those states.

2. It is often useful for a step in a trace to contain an explicit description of the knowledge state which occured before and after that step. This containment is described in Section 3.2 and its usage for adaptation is detailed in Section 6.4.

3. It is possible that the execution of a lower level task may create a binding for a parameter which contradicts the binding of that parameter at a higher level. Such contradictions are common, for example, in the context of recursion; if a task invokes itself, it normally does so on different parameter bindings. By explicitly maintaining the current knowledge state during execution of a task, it is possible to maintain the bindings within the state at that level while allowing lower level tasks to use their own bindings.

In principle, a knowledge state should contain *all* of the information contained within the agent at the moment that it is recorded. Such information would include not only various Loom concepts, relations, instances, etc., but all of the other sorts of data items within the agent. For example, a knowledge state involving the URL concept described Section 2.4.1 would involve not only the URL concept and its instances, but also all of the values of the text slots, which are simple LISP strings. Furthermore, some data items within an agent are likely to be more complex than strings (e.g., nested lists, arrays, functions) and many may not be referred to by any Loom instance. In addition, it is possible for both code and data for a REM agent to be encoded in a language other than LISP; such code and data would be accessed by REM (which does run in LISP) through an external function interface. Finding and storing all of this data would be enormously difficult, both in terms of the amount of programming effort required to handle all of the different kinds of information which can exist within a program, and in terms of the costs in time and memory space to store all of that data.

Fortunately, the three primary motivations for recording knowledge states do not really require that all information within an agent be stored. In particular, the first motivation only really demands that there be some representation, the second demands that this representation contain enough information to enable adaptation, and the third demands that it include bindings of parameters to values. Thus it is possible for knowledge states to actually be only a partial record of all of the knowledge in a system at a step in a reasoning process. One relatively obvious approach to knowledge states which would satisfy these criteria would be saving all of the knowledge encoded at Loom at each step. Since REM only formally represents the knowledge of an agent encoded in Loom, its reasoning about knowledge is almost entirely limited to reasoning about Loom knowledge. Thus knowledge states of this sort encode the most significant aspects of knowledge in REM (satisfying the motivation 1) and thus provide most of the information that one could conceivably want to use in adaptation (satisfying motivation 2). Furthermore, REM explicitly represents parameters and bindings in Loom, so this approach would also satisfy motivation 3. Loom provides an extremely convenient mechanism for doing exactly this: contexts. A context in Loom is a data structure which encapsulates a body of Loom knowledge. Loom allows a context to be a subcontext of one or more other contexts. The lower level

contexts inherit information from the higher level ones. New information in the subcontexts overrides the information in the parent contexts. Thus one could envision representing each knowledge state in the sequence of an execution as a subcontext of the knowledge state which it proceeded it.

In fact, during the early stages of the development of REM, Loom contexts were used as knowledge states in exactly this way. This approach proved very easy to implement and provided plenty of useful information for adaptation. Unfortunately, there was one overwhelming drawback: the computational cost. Because TMKL models can include loops, recursion, etc., an agent of any size can have an unlimited number of steps in the execution. In practice, agents with a few dozen tasks and methods frequently involve hundreds of steps for a single execution; by the end of that execution, contexts are nested hundreds of levels deep. Although no formal experiments were conducted at that early stage of development, it was obvious from even casual observation that this approach to representing knowledge states was prohibitively expensive.

The approach which has ultimately been taken in REM is much lighter weight. REM has a concept, knowledge state, with one associated relation, knowledge-state>binds. This relation is a trinary relation which maps knowledge states and parameters to values. In other words, a knowledge state in REM contains only parameter bindings;[1] all other knowledge is not recorded in the knowledge state and thus is not available within a trace. The parameter bindings are a fraction of an entire Loom context. However, they do directly satisfy motivation 3. Furthermore, because parameters act as the bridge from knowledge to tasks and methods, most of the reasoning that one might want to do in adapting tasks and methods using knowledge states is based around parameters; to the extent that slots in a task (e.g., in the makes state of a task) refer to information other than parameter values, they typically refer to static (and thus state independent) information. Thus this perspective on knowledge states provides most of the sorts of information that one would want to represent in a model of REM and reason about during adaptation, thereby providing a reasonable answer to motivations 1 and 2. In principle, it is conceivable that some sorts of adaptation could require additional information than what is provided by this approach. However, the adaptation mechanisms currently in REM are able to operate with knowledge states of this sort. Thus the current implementation of REM suggests that encoding knowledge states as a set of bindings between parameters and values constitutes a fair compromise between power and efficiency.

3.2 Traces

A trace is a representation of a particular execution of an agent. The primary purpose of traces in REM is the facilitation of failure-driven adaptation; when a failure is detected during execution or is asserted by the user after execution (in the form of feedback), REM uses the trace to guide the identification of pieces of the model which may need changing and to provide information which can be used to enable those changes. Information about the way that traces are produced during execution is presented in Chapter 5. A description of how traces are used for adaptation is provided in Section 6.4. The text here focuses on the content of traces.

Traces are organized hierarchically, like TMKL models are. The top level of a trace of a TMKL agent is a task trace for the main task that was invoked. If a method was invoked for the main task, the task trace contains a method trace. The method trace, in turn, contains transition and reasoning state traces, and the reasoning state traces contain task traces for the subtasks. Unlike TMKL models, however, there are never any branches, loops, or alternatives in a trace; the trace only records the path which was taken during a particular execution. In some cases a trace may be much smaller than the corresponding model since only certain tasks and methods within the agent were performed. In other cases, however, a trace may be much larger than a model because a model only contains one representation of each task and method while a trace contains a separate representation of each time a task or method was invoked; for example, if there is a loop within an agent, the model will have a single representation of each step in the loop while the trace will represent each step as many times as the loop was executed.

Table 13 provides a general schema for a trace. This general schema is appropriate to all elements of any given trace, i.e., any piece of a trace is, itself, a trace. For example, the trace for a non-primitive task contains the trace for the method that was invoked for it. At the bottom level of the trace hierarchy are simple atomic traces of primitive tasks. The general schema for a trace contains two slots: one for failure annotations and one for a feedback annotation. These annotations are described in Section 3.3.

[1] More formally, knowledge states are encoded in such a way that given a knowledge state, ?k, the Loom query (retrieve (?p ?v) (knowledge-state>binds ?k ?p ?v)) returns a LISP association list such that each association corresponds to a binding of a parameter to a value at the point in the execution in which ?k was recorded.

Concept **Trace**		
Slot	*Type*	*Quantity*
failures	trace failure	0+
feedback	feedback	0-1

Table 13: General schema for an element of a trace

Concept **Task Trace** is-a Trace		
Slot	*Type*	*Quantity*
task	task	1
invoked-mmethod-trace	mmethod-trace	0-1
input-knowledge-state	knowledge-state	1
output-knowledge-state	knowledge-state	1

Table 14: Schema for a task element of trace

Table 14 presents the schema for a task trace. Since a task trace is a subconcept of the general trace concept, it inherits the two slots from that schema (failures and feedback). The task trace schema also contains four additional slots. The task slot simply refers to the task in the TMKL model for which the trace was built. The invoked-mmethod-trace slot refers to a trace for the method (if any) which was invoked for that task. Finally, the input-knowledge-state and output-knowledge-state slots contain the knowledge states which appeared before and after the task was executed; this information allows later analysis to reason about what the task did (e.g., to contrast with what the task should have done).

Tables 15, 16, and 17 show the schemas for method traces, transition traces, and reasoning state traces. Each of these types of traces inherits the two kinds of annotations from the general trace concept. In addition, each of the sorts of traces refers to a TMKL model element of the type that it is a trace for; thus each piece of a trace refers directly to a corresponding piece of the model. Lastly, each kind of trace refers to the next kind of trace in the execution. A method trace refers to the first transition trace for that method. A transition trace refers to the next reasoning state trace (except for traces of terminal transitions, which have no next trace). A reasoning state trace refers to both the task trace for that state's subtask and the next transition which was selected from that state (if any). Note that method, transition, and reasoning state traces do not contain any information about knowledge states; however, this information is relatively easy to infer from the knowledge states in the task traces. For example, the knowledge state at the beginning of a method is simply the input knowledge state for the task that invoked the method.

The combination of the different varieties of traces allows an overall trace of a main task to be composed of traces for lower level elements. The task traces at the bottom hierarchy refer to the primitive tasks that were executed and the input and output knowledge states for those primitives. Higher level task traces describe the input and output knowledge states for larger portions of the agent. Additional sorts of traces (method, reasoning state, and transition) provide additional details about what pieces of the model were executed and in what order. The various traces together provide a description of the process and results of the execution of a TMKL agent.

Concept **Method Trace** is-a Trace		
Slot	*Type*	*Quantity*
mmethod	method	1
start-trace	transition-trace	1

Table 15: Schema for a method element of trace

Concept **Transition Trace** is-a Trace		
Slot	*Type*	*Quantity*
transition	transition	1
next-reasoning-state-trace	reasoning state trace	0-1

Table 16: Schema for a transition element of trace

Concept **Reasoning State Trace** is-a Trace		
Slot	*Type*	*Quantity*
reasoning-state	reasoning state	1
subtask-trace	task-trace	1
next-transition-trace	transition-trace	0-1

Table 17: Schema for a reasoning state element of trace

3.3 Trace Annotations

Because a trace is a record of a specific execution of an agent, one way to provide additional information about some execution is to place annotations on the trace. Two general classes of trace annotations are failures and feedback: the former represent externally supplied information about the execution and the latter represent points in the trace where some problem occured.[2] Note that feedback is generally supplied directly by the user while failures are normally automatically detected by the algorithms in REM (either during execution or during the blame assignment portions of adaptation; see Chapters 5 and 6 respectively). The following two subsections describe these two kinds of trace annotations.

3.3.1 Feedback

Feedback is information provided by a user about a particular execution of an agent. Table 18 provides the schema for feedback. There is one slot for a piece of feedback: a logical assertion, using the same syntax as logical assertions within a TMKL model (i.e., in the asserts slot of a task). Note that the semantics of placing an assertion within a piece of feedback is subtly different than simply declaring it to be true. For example, consider the following situation in the light bulb domain. There is a task insert-bulb which should insert an arbitrary bulb into a socket; specifically, it should transform a state in which the bulb is in the hand into a state in which the bulb is in the socket. Furthermore, the agent has a particularly large bulb named big-bulb and knows of two knowledge states involving that bulb: big-bulb-in-hand (in which it knows that big-bulb is in the hand) and big-bulb-in-socket (in which it knows that big-bulb is in the socket). Consider the following Loom statement:

```
(tell (should-transform
        insert-bulb big-bulb-in-hand big-bulb-in-socket))
```

[2]No such annotations exist in SIRRINE; the handling of failures and feedback in SIRRINE is much simpler and more *ad hoc*. The integration of failure and feedback information with the trace is one of the novel contributions of REM.

Concept **Feedback**		
Slot	*Type*	*Quantity*
assertion	logical assertion	1

Table 18: Schema for feedback

This is a simple, direct assertion within Loom. It simply states that insert-bulb should transform big-bulb-in-hand into big-bulb-in-socket. More precisely, it indicates that should-transform[3] holds over these items. Contrast that with the following Loom statement:

```
(tell (:create feedback-1 feedback)
      (trace>feedback trace-1 feedback-1)
      (feedback>assertion feedback-1
        '(should-transform
           insert-bulb big-bulb-in-hand big-bulb-in-socket)))
```

This statement assumes that there was already some existing trace trace-1. The statement indicates that the problem with trace-1 is that insert-bulb should transform big-bulb-in-hand into big-bulb-in-socket. One of the differences between that statement and the former statement is that the feedback statement explicitly indicates that there was a problem with trace-1; this is particularly important if no failures were detected during the execution recorded in trace-1 because if there are no failures and no feedback for a trace then REM assumes that there is no reason to adapt the agent, and thus does not do so (as indicated in Chapter 1). The other important distinction between the two example statements above is that the second statement suggests that one way to fix the problem that was encountered with the trace is to address the should-transform assertion. Thus when REM begins to analyze the trace (see Section 6.4) it knows that it should be looking for contradictions with the assertion in the feedback. In this case, it looks for a portion of the trace in which insert-bulb received big-bulb-in-hand as input but did not produce big-bulb-in-socket as output. In principle, a system could search a trace for contradictions with every piece of knowledge that it has. However, this approach could potentially be overwhelmingly costly. In practice, REM only uses the knowledge in the feedback for this sort of analysis; i.e., it will be able to use the assertion in the second `tell` statement above, but not the assertion in the first one.

In principle, any sort of assertion can go into a piece of feedback. However, the mechanism for interpreting feedback in REM is currently only capable of processing one sort of assertion: a single clause whose relation is should-transform. In principle, it would be useful to address feedback that includes a broader class of relations and even complex assertions with multiple relations. This issue is discussed in more detail in Section 9.1.2.

3.3.2 Failures

When REM detects that some failure has occured for a particular execution of an agent, a failure annotation is attached to the trace that was generated in that execution. Note that this detection can occur either during the execution itself or during some later analysis. REM is capable of both of these sorts of failure detection. A description of how failures are detected during execution occurs in Section 5.3 and a description of how failures are detected during later analysis occurs in Section 6.4. Section 6.4 also describes how failure annotations from both of these sources are used to facilitate adaptation.

REM represents failure annotations using the trace-failure[4] concept and a number of subconcepts. The general trace-failure concept has no slots; however, some of the subconcepts do provide slots in which to supply additional information about the failure; e.g., the missing-inputs failure (below) has a slot to state which inputs were missing.

There are four simple failure types (i.e., subconcepts of trace-failure which have no slots): given-fails, makes-fails, no-applicable-mmethod, and no-applicable-transition. Each one of these types of failures is detected during execution. The first indicates that the given condition for a task was not met at the start of execution of that task. The second indicates that the makes condition for a task was not met at the end of execution of that task. The third indicates that there were no methods which were available for a particular task (i.e., which had their provided conditions met). Finally, the fourth indicates that there were no transitions which were available for a particular reasoning state within a method.

In addition to those four simple failure types, there is one more kind of failure which is detected during execution: missing-inputs. This failure is detected when a task has one or more inputs which are not included in the knowledge

[3]The should-transform relation is defined within REM; it is a trinary relation which denotes that a task should transform one knowledge state to another, i.e., that if the first knowledge state is provided as input then the second knowledge state should be produced as output. The primary purpose of this relation is for feedback (i.e., to contrast the what the task should have done with what it actually did). For more on the use of this particular relation in adaptation see Sections 6.4.2 and 9.1.2.

[4]As a minor aside, there is a departure here from the convention used in this chapter and the previous one of describing concepts using normal text and only providing technical terms for slots, relations, etc. This departure is motivated by the fact that the trace-failure has a complex hierarchy of subconcepts, all named after the circumstances they denote. It seems easier to keep track of the different kinds of failure annotations and what they are for if they are clearly presented as technical terms.

Concept **missing-inputs** is-a trace-failure		
Slot	*Type*	*Quantity*
inputs	parameter	1+

Table 19: Schema for missing-inputs

Concept **feedback-trace-failure** is-a trace-failure		
Slot	*Type*	*Quantity*
feedback	feedback	1
desired-input	knowledge-state	1
desired-output	knowledge-state	1

Table 20: Schema for feedback-trace-failure

state in which the task was to be executed. Table 19 provides the schema for this subconcept of trace-failure. It has one slot, inputs, which lists the input parameters which did not have values in the input knowledge state.

The types of failures that REM detects during analysis after execution are all based on feedback; REM searches the trace for elements in which what did happen conflicts with what the feedback says should have happened. Table 20 provides a general schema for failures based on traces. REM can identify two specific types of failures based on traces; there are two corresponding subconcepts of feedback-trace-failure: directly-contradicts-feedback and may-contradict-feedback. The former denotes a situation in which the feedback asserts something which unambiguously conflicts with the trace; for example, if the feedback says that a certain task should have produced a certain value in a certain situation and it produced some other value, a directly-contradicts-feedback annotation is asserted. The other type of failure based on feedback indicates that there is a possible conflict between the feedback and the trace; for example, if the feedback indicates that some high-level task should have produced a value and some subtask of that task which could have produced that value didn't, then this may be an indication that the subtask should be modified, and a may-contradict-feedback annotation is asserted. Note that the ability to detect may-contradict-feedback failures is a particularly vital one because it is generally expected that most feedback provided by a user will be about high-level tasks (especially the main task which the user had asked to be executed), but that most modifications to an agent will occur at a lower level; thus it is essential for REM to be able to diagnose high-level feedback as potentially indicating failures of lower-level components. The PDF viewing problem described in Section 6.4 is one example of this sort.

Recall that traces of complex tasks and methods are composed of traces of subtasks, methods for subtasks, etc. Failure annotations are attached directly to the specific trace for which the failure occured. For example, a given-fails annotation is attached to the task trace for the part of the execution in which a task's given condition was not met. As noted in Section 3.2, each of these traces refers directly to the portion of the model which it records, so a failure annotation is tied (through the trace that it annotates) to some specific piece of a model. Thus in addition to providing information about the type of failure and the knowledge involved in the failure, the annotation also indicates where the failure occured. In other words, a failure annotation serves four major purposes:

- To denote that a failure has occured during execution.
- To indicate the type of failure that occured.
- To localize the failure to a particular potion on the TMKL model.
- To provide additional type-specific information about the failure.

The failure annotations are developed during the blame assignment portions of both execution and later analysis. They provide essential guidance to the model repair mechanisms in describing what to repair and how to make those repairs.

Chapter 4

Example Agents

The model-based reflection techniques described in this dissertation have been applied to a diverse variety of agents. In this chapter, the most significant of these agents are presented in detail. In part, these detailed descriptions are provided because the agents and models themselves are products of this research. However, the primary reason why descriptions of the agents appear in this dissertation is to provide illustrative examples. Chapters 5 and 6 use these examples to help describe how the execution and adaptation processes work. Chapter 7 then presents experimental results using these example agents.

4.1 ADDAM

ADDAM [Goel et al., 1997a] is a physical device disassembly agent. Note that ADDAM was not created as part of this research, and the original ADDAM system did not contain any representation of itself and thus did not have any ability to reflect on its own reasoning. The work on ADDAM was done for this dissertation research consists of three parts: (i) an analysis of the existing ADDAM system in terms of the tasks, methods, and knowledge that it uses, (ii) the development of a TMKL model of ADDAM integrated with the existing ADDAM code for accomplishing pieces of the disassembly process, and (iii) a series of experiments involving the execution and adaptation of ADDAM in REM. Note that these three items are closely interrelated; the analysis supported the construction of the model which enabled the experiments. In this section, we describe ADDAM, itself, focusing on the TMKL model as a formal description of the agent. Information about how ADDAM is used in REM appears many places in the remainder of this thesis (particularly Chapters 5, 6, and 7).

The original motivation for the ADDAM project was the fact that the planning of disassembly processes is both a challenging and important problem. There is an ever increasing variety of complex physical artifacts which are commonly used for only brief periods of time and then destroyed. Consider, for example, a disposable pocket camera. When such a camera is used up, it could simply be tossed into a trash can and eventually wind up in a landfill. However, if there were a cheap and efficient way to automatically disassemble a disposable camera, then some components could be directly reused and the raw materials in others could be recycled. ADDAM is a proof-of-concept system that addresses this problem in simulation.

There are two primary reasons why disassembly is difficult task to automate:

- The entire sequence of physical movements for disassembling a device is extremely long and complex.
- Combining arbitrary actions to form a sequence which satisfies an arbitrary goal is computationally expensive.

ADDAM addresses the first of those issues by reasoning about devices and actions at a relatively high level; for example, ADDAM represents the entire process of unscrewing two components as a single atomic action, even though this process is actually composed of many individual movements to get the screwdriver into the screw, turn the screwdriver, pull the screw out, etc. Of course, to actually be useful for automation, a disassembly plan does need to ultimately be translated into robotic movements. ADDAM has a separate module called Low-Level ADDAM (LLA) which addresses this issue. LLA translates individual actions from the high level plan into combinations of movements for a detailed robot simulator. Note, however, that the experiments of ADDAM within REM have not involved the LLA module at all because LLA is severely limited and inconvenient to use, is not part of the main ADDAM system, and is not particularly relevant to the issues in the REM / ADDAM experiments. LLA is mentioned here only because the fact that it exists is a key part of the theory behind ADDAM; the issue of the complexity of planning low level movements is resolved by first developing a plan at a high level and only translating that plan into physical movements when necessary.

However, reasoning about high-level actions does not completely resolve the second issue: the cost of combining actions to form a goal. Approaches to this problem which use relatively little knowledge can sometimes be extremely expensive, even when reasoning about a fairly small number of actions and objects. Some fairly simple devices are prohibitively expensive to disassemble without advance knowledge, even when represented and reasoned about at the level of detail in ADDAM (as demonstrated by experiments described in Chapter 7). The approach that ADDAM takes to this problem is case-based reasoning; ADDAM adapts old plans for assembling devices into new plans for disassembling similar devices. This approach is particularly well-suited to environments in which there is a relatively small variety of types of devices which need to be disassembled but that variety changes gradually over time. For example, a disposable camera company is likely to have a fairly small number of models of cameras. It seems feasible to simply develop disassembly plans for all of these models by hand. However, over time, new models of cameras are developed. Constantly maintaining the library of disassembly plans by hand may be very costly. ADDAM addresses this challenge by making it possible to modify disassembly plans for older models into disassembly plans for newer models.

4.1.1 ADDAM knowledge

The representation of devices in ADDAM describes not only individual components and connections but also more abstract subassemblies that are composed of those components and together comprise the device. The combination of components, connections, and subassemblies forms a topological description of a device. For example, ADDAM has a representation of a computer which includes a storage subsystem; that subsystem is, in turn, composed of components such as a hard drive, a controller card, and a cable. ADDAM allows device hierarchies to be arbitrarily deep, i.e., a device can be composed of subsystems which are composed of subsystems, etc.

Disassembly plans in ADDAM are also hierarchical in nature; a disassembly plan is based on the topology of the device that it affects. For example, there is a node in the computer disassembly plan which involves disassembling the storage subsystem. That node has children which involve disconnecting and removing the various components of the subsystem. The primary benefit of these hierarchical plans in ADDAM is that they allow plans for entire subsystems to be reused. In the computer example, when a new computer is presented which has two storage subsystems, ADDAM is able to take the entire portion of the plan for disassembling the storage subsystem in the original computer and reuse it twice. The process for adapting plans (detailed in the next section) is organized around the hierarchical structure of the plans and devices, not around the order in which the actions are to occur. Plans in ADDAM are partially ordered, i.e., instead of having a complete specification of the order in which actions occur, a plan has an arbitrary number of ordering dependencies which state that one particular plan node must be resolved before another.

One example of an object which ADDAM is able to disassemble is a hypothetical layered roof design involving of a variable number of boards. Figure 18 depicts this roof. The design is very simple, consisting only of boards and screws. However, the configuration of the roof is such that the placement of each new board obstructs the ability to screw together the previous boards so the assembly must be constructed in a precise order, i.e., place two boards, screw them together, and then repeatedly place a board and screw it to the previous board until all boards are placed.

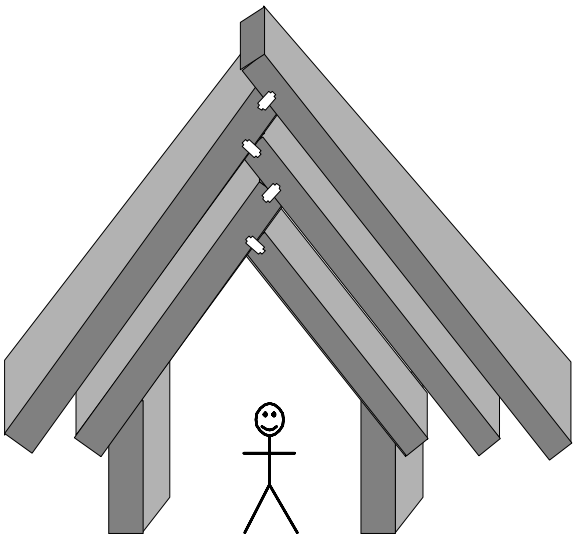


Figure 18: A sketch of the layered roof. Note the fact that the higher level layers are stacked on top of the lower level ones, blocking access to them. This makes it necessary to assemble or disassemble all of the components in a particular order.

4.1.2 ADDAM process

Figure 19 shows the tasks and methods of ADDAM. The top level task of ADDAM is Disassemble. This task is implemented by ADDAM's process of planning and then executing disassembly. Planning in ADDAM is adaptive; it consists of taking an existing plan for disassembling a similar device and adapting it into a plan for the new device. ADDAM's method for planning, Topology-Based Plan Adaptation, is divided into three subtasks: Match Toplogies, Make Plan Hierarchy, and Map Dependencies. Match Topologies involves stepping through the topological descriptions of the devices and forming mappings between elements of the old device and elements of the new device. Make Plan Hierarchy then uses these mappings to translate the structure of the old plan into a structure for the new plan. The method for this task operates hierarchically. Its first task, Select Plan Base Top, extracts the top node of the plan and treats that top node as a subplan. The next task, Make Subplan Hierarchy, recursively adapts an old subplan hierarchy into a new subplan hierarchy while maintaining a set of mappings from nodes in the old plan to the new plan nodes. Note that Make Subplan Hierarchy is the most complex portion of ADDAM and is described separately later in this section. After the top-level subplan is complete, the Encapsulate Target Plan task declares that the top-level subplan hierarchy is the plan hierarchy for the original disassembly problem.

After the plan hierarchy is built, the Map Dependencies task is executed; this task involves imposing ordering dependencies (i.e., requirements that one be executed before another) on steps in the plan. The first part of this process, List Target Dependencies, constructs a list of dependencies based on the mappings between nodes in the old plan and nodes in the new plan which were recorded during the Make Subplan Hierarchy process. Whenever the old plan has a dependency between two plan nodes and the new plan hierarchy has analogs to both of those plan nodes, then the dependency in the old plan needs to be included in the new plan. After the list is built, it is stepped through one dependency at a time; each individual dependency is selected (Select Dependency) and then added into the plan (Assert Dependency).

Once the ordering dependencies have been added to the plan, the adaptive planning process is over. The next step in the process is the Execute Plan task. The first step in this process, Sequentialize Plan, involves taking the hierarchical, partially ordered plan that was produced by the adaptation mechanism and extracting from it a simple sequence of actions which is consistent with the ordering dependencies. After that, execution is very simple; actions are taken from that sequence (Select Next Action) and executed (Execute Action) until the end of the sequence is reached.

Recall that the most complex portion of ADDAM is the Make Subplan Hierarchy task. The task is given a current plan node (i.e., a plan node in the old plan hierarchy) and it produces one or more plan nodes for equivalent structures in the topology of the new device. For example, when adapting a plan for a computer with one storage subsystem

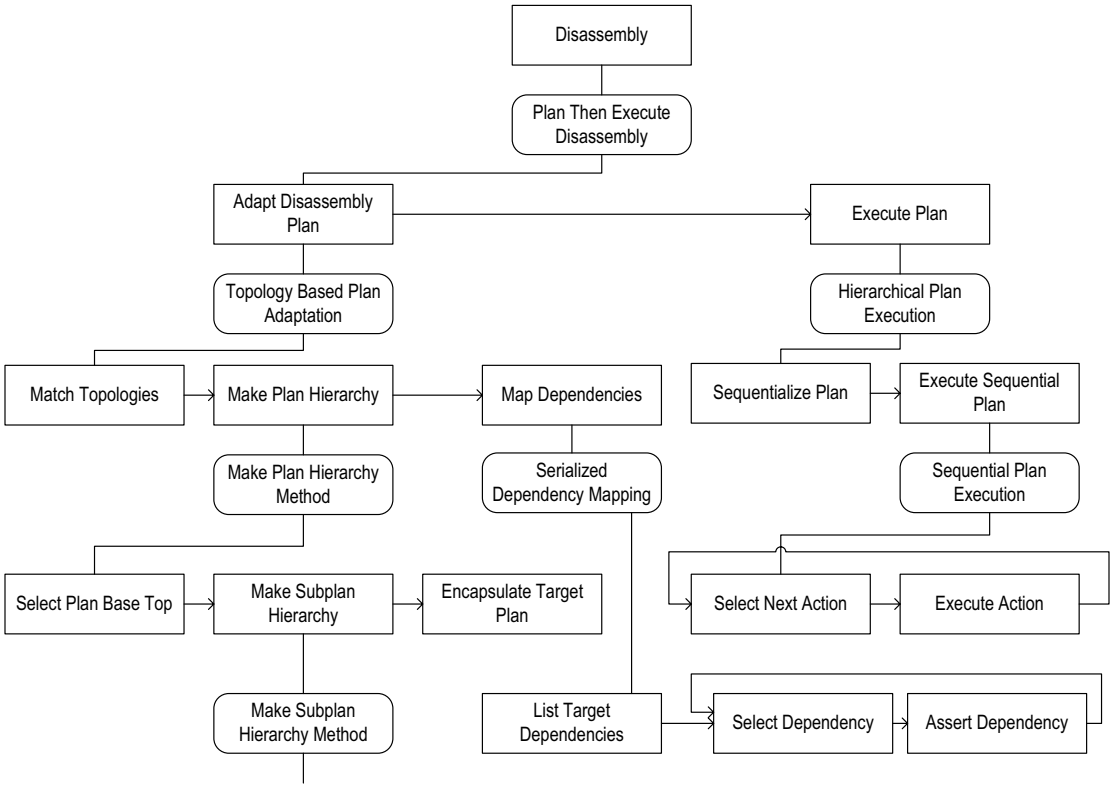


Figure 19: The tasks and methods of ADDAM. The subtasks of the Make Subplan Hierarchy Method are not shown here; they appear in Figure 20.

to a new computer with two storage subsystems, one invocation of the Make Subplan Hierarchy task will receive the plan node for the storage subsystem in the old computer and will produce plan nodes for both storage subsystems in the new computer (and it will recursively construct the nodes for their children).

Table 20 shows the task structure for Make Subplan Hierarchy. The first step in its method is Find Equivalent Topology Nodes. This task finds nodes in the new topology which are equivalent to the node in the old topology which is manipulated by the current plan node (e.g., it would find the storage subsystems in the new topology). Next, the Make Equivalent Plan Nodes subtask steps through the equivalent nodes in the new topology one at a time, making new plan nodes for them by adapting the current plan node (e.g., taking the plan node for the old storage subsystem and transferring it to each new storage subsystem separately). This process involves three steps which occur in a loop: selecting one of the topology nodes, making the new plan node, and adding the new plan node to the task. After Make Equivalent Plan Nodes, the next step is Make Plan Node Mappings. This task involves recording that all of the new plan nodes which were just created are analogous to the current plan node; this record of mappings is used later in ADDAM to guide the dependency mapping process.

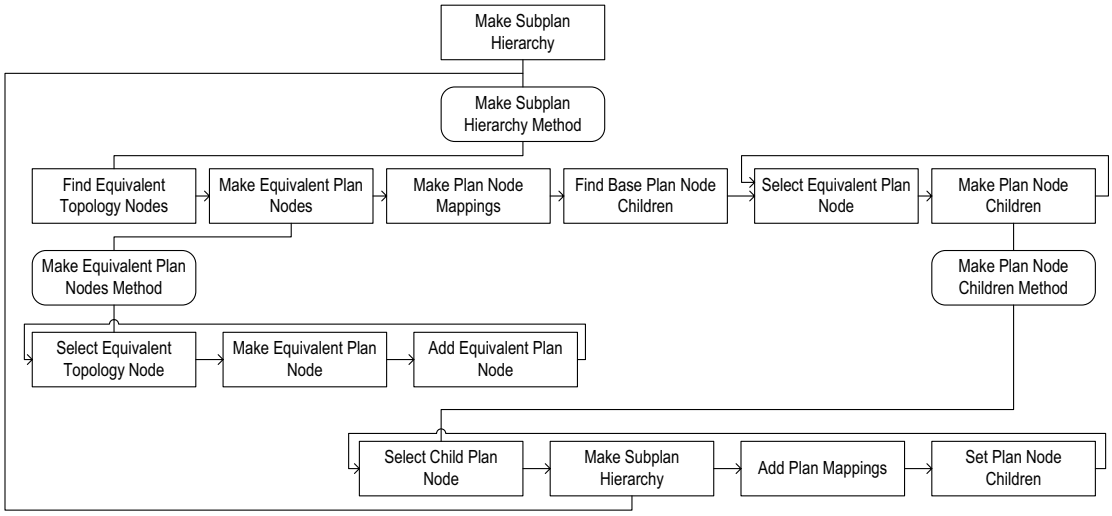


Figure 20: The Make Subplan Hierarchy task, its method, subtasks, etc.

The next three subtasks involve building the new plan nodes for the children of the current plan node. Find Base Plan Node Children extracts all of the children from the current plan node (e.g., the disassembly actions for the components of the old storage subsystem); note that if the current plan node is a primitive action then there will be no children and the loop that follows will be executed zero times. In this loop, Select Equivalent Plan Node first chooses one of the newly built plan nodes (e.g., the plan node for the first storage subsystem in the new computer). Then Make Plan Node Children builds new children for the the new plan node that was just selected. These last two steps (selecting a new node and making children for it) repeat for each of the newly built plan nodes (e.g., the nodes for the two storage subsystems).

The Make Plan Node Children task, itself, has several subtasks. The first of these, Select Child Plan Node, chooses one of the children which were produced by Find Base Plan Node Children (e.g., the node in the old plan which involved removing the hard drive controller card). After that, the Make Subplan Hierarchy task is recursively invoked to build new plan nodes for that existing plan node. Next, Add Plan Mappings combines the mappings which were built in the recursive call to those which were built at the current level (so the set of mappings is built up incrementally). Finally, the Set Plan Node Children task involves modifying the plan node which had been selected by Select Equivalent Plan Node to include among its children the new plan nodes produced by the recursive call to Make Subplan Hierarchy; for example, if the equivalent plan node is the node for the first storage subsystem in the new computer and a new node produced by the recursive call is a node for the hard drive controller card, the latter node is asserted to be a child of the former. The steps of the Make Plan Node Children task are repeated until all of the new plan nodes have all of their children produced.

The overall effect of the ADDAM process is the simulated disassembly of the device. The process begins by analyzing the topologies of the old and new devices. After that, it builds a new plan hierarchy by transferring the old plan hierarchy for the related portions of the topologies. Next, it imposes ordering dependencies on the new

plan, also by transferal from the old plan. Finally, ADDAM executes the plan, causing the simulated device to be disassembled.

4.2 Web Browsing Agent

One agent which has been the subject of experiments in both SIRRINE and REM is a web browsing agent. This agent is intended to initially imitate the behavior of the Mosaic web browser and to intelligently develop new capabilities. It was built by imitating both the functionality and structure of Mosaic for X Windows, version 2.4. Our initial (pre-adaptation) model of the web browser is based on an architectural analysis of Mosaic 2.4; this analysis was developed using the SAAM methodology [Abowd et al., 1997].

The web browsing agent is a non-functioning mock-up. It steps through the process of web-browsing but does not actually access the web or display documents. One reason why such a system could be useful is in support of semi-automated software engineering: given a challenge which would require that the actual software system (in this case Mosaic 2.4) be modified, a software engineer can see how SIRRINE or REM would address that challenge for the mock-up agent and then use the changes made as a potential redesign for the actual system.

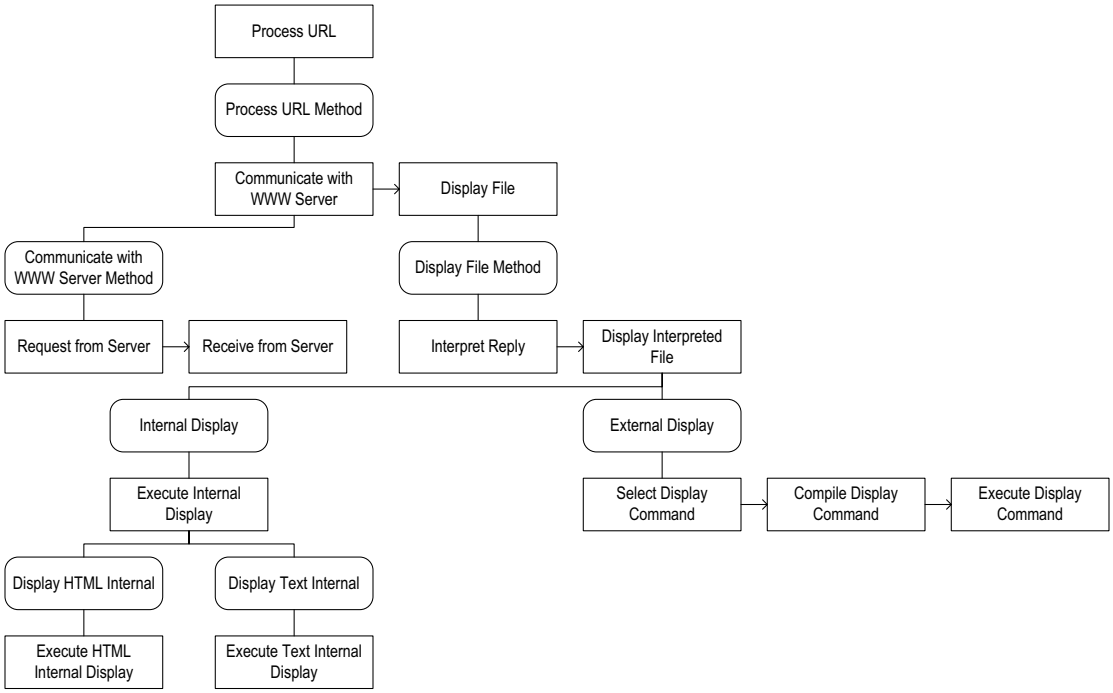


Figure 21: The tasks and methods of the web browsing agent.

Figure 21 describes the tasks and methods of the web browsing agent.[1] The top level task in the web browser is Process URL. This task takes as input a URL; its goal is to have the document at the location indicated by the URL be displayed on the local terminal. The method for this task has two steps. The first, Communicate with WWW Server, gets the document from the web by requesting it and receiving it. Then the Display File task presents this document on the local host. The first step in that process, Interpret Reply, extracts a MIME tag from the response generated by the network. Then, if the MIME tag corresponds to a type which can be displayed internally, e.g., HTML, the system does so (in the Internal Display method). Otherwise, the External Display method is invoked. This method involves first selecting a display program from a precompiled table of MIME types and display programs (Select Display Command), and then constructing a command to send to the system shell which invokes that program on the document (Compile Display Command), and finally sending this command to the system shell, causing the document to be displayed (Execute Display Command). Once the document has been displayed, either internally or externally, the overall Process URL task is complete.

[1]The figure shows the tasks and methods of the web browser, as it is encoded in REM; the web browsing agent in SIRRINE is essentially the same but has a slightly less detailed model.

4.3 Monkey and Tower Problem

The monkey and tower problem is a relatively simple "toy" problem which has been addressed in REM primarily for illustrative purposes. It involves a combination of two traditional illustrative examples in Artificial Intelligence: the Monkey and Bananas problem and the Tower of Hanoi problem. In this combination, the agent acts as a simulated monkey that is trying to obtain some bananas which are hanging from the ceiling. The bananas are too high in the air for the monkey to reach, but the monkey does have a set of disks that can be stacked in such a way that they form a tower which the monkey can climb and obtain the bananas. The disks are all of different sizes and they are balanced in such a way that no disk can be stacked on a smaller disk. There are three locations where the disks can be placed; they all begin stacked on one of these locations (which is not the location where they need to be to get the bananas). The monkey can move one disk at a time.

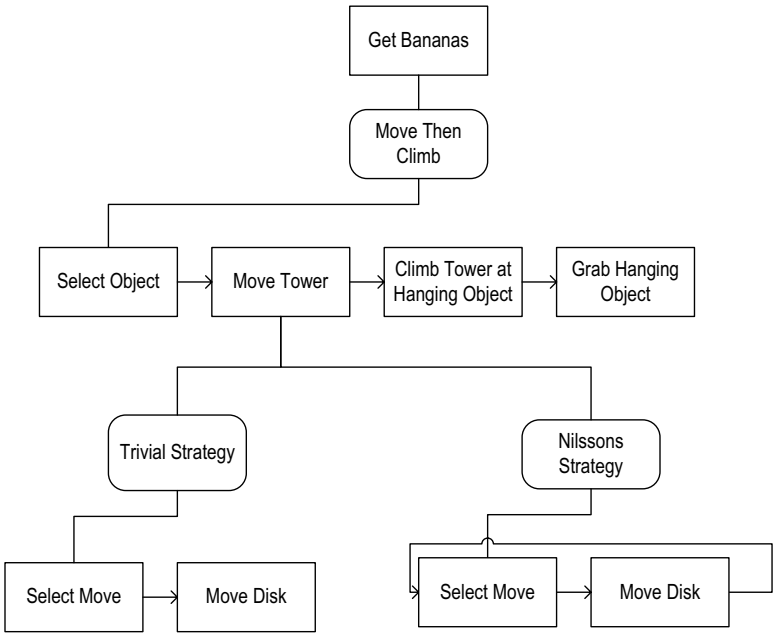


Figure 22: The tasks and methods of the monkey and tower problem agent.

Figure 22 shows the tasks and methods of the example agent in REM which addresses this problem. The overall task, Get Bananas, is decomposed into four subtasks: Select Object, Move Tower, Climb Tower at Hanging Object, and Grab Hanging Object. The first task outputs an object to find; specifically, it always finds bananas hanging over the end location. The second task moves the disks from the start location to the end location, using the intermediate location if necessary. It has two separate methods: Trivial Strategy and Nilsson's Strategy. The former has a provided condition which must hold to be executed: that there be one disk. That strategy simply moves the disk to the destination. Nilsson's Strategy, on the other hand, has no provided condition, i.e., it can be used for any number of disks. It is a generalization of the Tower of Hanoi strategy in [Nilsson, 1998, errata, p. 4].[2] The process is a simple, iterative one; there is a fixed rule which determines the next move to make at each step. The rule states that at each step the movement selected is the one which moves the largest disk possible under the rules of the problem and the restriction that given n disks, a disk numbered m can only move clockwise if the parity of m matches the parity of n and can only move counter-clockwise if the parity of m does not match the parity of n. Note that the details of the rule are not particularly important here since the rule is implemented in a procedure in this agent (and thus exists at a level beneath which model-based adaptation can reason). What is significant is the fact that there is a rule which selects moves, and the movement process involves iteratively selecting and executing moves until the tower is at the final location.

The two methods for the Move Tower task are redundant; the behavior of Nilsson's Strategy when there is only one disk is identical to the behavior of Trivial Strategy. Both methods are included to allow experiments which involve one or the other. For example, one of the problems in Section 6.4 involves executing a variation of the monkey and

[2]That text only describes the strategy for 3 disks. Since no citation is given, it is presumably the original source.

tower agent in which Nilsson's Strategy is removed[3]; this agent is able to behave correctly when only one disk is present, but initially fails when more than one disk is present. That failure leads to model-based adaptation which eventually leads to a correct solution.

4.4 Other Examples

The examples in the previous subsections are the ones which have been experimented on most thoroughly in the course of this research. However, there have been a number of other examples which have considered. Most notable among these other examples are systems which have been encoded in SIRRINE but not in REM. In particular, some important example agents from SIRRINE include:

Meeting Scheduling Agent : This is a moderately complex agent which was built for SIRRINE. It takes as input a set of schedules plus a length of time and searches for a time slot of the given length which fits into the given schedules. The mechanism it uses for finding a slot is a generate-and-test strategy in which the agent repeatedly produces a slot and then checks to see if that slot fits into all of the schedules. Experiments with the meeting scheduler have focused on the issue of constraints (e.g., what sorts of slots the agent is able to produce). For more about these experiments, see Section 7.1.5.

Tic Tac Toe Game : This agent was built for SIRRINE; it selects moves for the simple children's game, Tic Tac Toe. The agent used a preliminary version of the reinforcement learning mechanism which is found in REM. This project was almost entirely unsuccessful. Because the agent made all of its decisions using reinforcement learning, it proved to be overwhelmingly expensive to run; specifically, even after days of execution, it never seemed to perform better than chance. The failure of this agent provided a preliminary motivation for the guideline within the REM project of trying to limit reinforcement learning to a small number of localized decisions. Note that the Tic Tac Toe agent was the only project in SIRRINE which did any reinforcement learning; because it was unsuccessful, the reinforcement learning component of SIRRINE was omitted from the final released version. However, the work on this agent was significant in that the lessons learned from it helped to motivate the decision making component in REM (described in Section 5.2).

Kritik : Kritik [Goel et al., 1997b] is a case-based system which performs autonomous design of physical devices using functional models. A TMK model of Kritik was developed for the Interactive Kritik [Goel et al., 1996], a system which presents graphical explanations of Kritik's reasoning processes and results. This model has been translated into SIRRINE. The model runs within SIRRINE, but no experiments with adaptation of the model have been run. The primary benefit of this translation has been an exploration of the consistency between the version of TMK in SIRRINE and earlier versions. The results suggest that the SIRRINE version of TMK can effectively represent a particularly complex earlier TMK system.

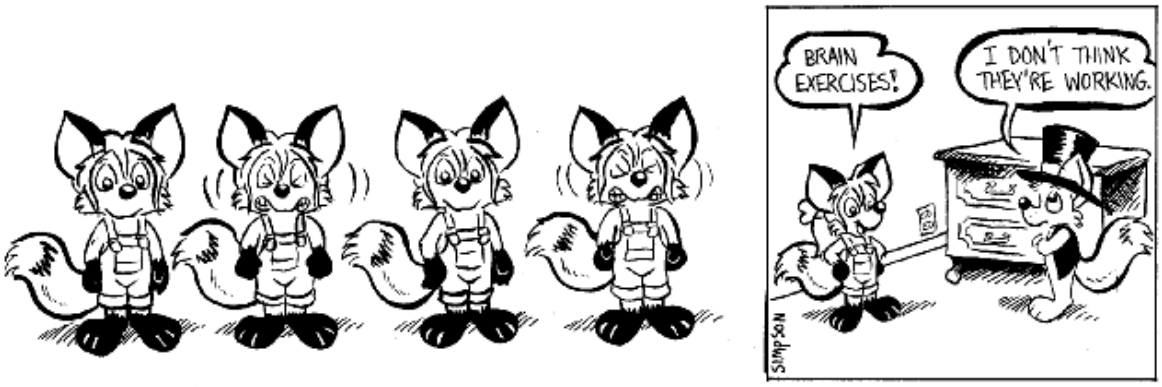
Employee Time Card System : This system is described in [Allemang, 1997], which describes the ZD language. ZD is a functional modeling language for representing software systems; more about the relationship between TMKL and ZD appears in Section 8.4. As with Kritik, no work on adaptation of this model has been done in SIRRINE. The primary purpose of the translation of the Employee Time Card system was the exploration of issues in language compatibility and interoperability. The fact that the model was able to be translated from ZD to TMK without enormous restructuring suggests that there is some fundamental compatibility between the two languages. In addition, the translation also revealed some subtle differences between the two languages; this motivated some of the language design issues which were considered during the development of TMKL (as explored in [Murdock, 1998a]). The Employee Time Card system was also used in some experiments involving translation into ACME [Garlan et al., 1997]; a language for interoperation in the context of Architectural Description Languages. This work has helped to clarify the relationship between functional modeling and software architectures; this relationship is explored in more detail in Chapter 8.

The agents presented in this chapter are the key experimental examples which have been considered in this research. Chapters 5 and 6 use these examples in describing the general processes by which these agents are executed and adapted. The combination of these examples and processes are the basis for the experiments presented in Chapter 7. Those results lead to the more abstract analysis which is found in the remaining chapters of this dissertation.

[3]Incidentally, the agent with Nilsson's Strategy removed constitutes a more realistic monkey, i.e., one who knows how to move a disk but who has not read the errata to [Nilsson, 1998]. This added realism is merely a coincidence, however. No version of the monkey and tower agent is intended to be a serious model of primate cognition.

Chapter 5

Execution



One essential function of a reasoning shell is the execution of agents which are encoded in the language of the shell. Most aspects of the execution of TMKL agents are relatively straightforward. When executing a non-primitive task, REM selects a method for that task. When executing a method, REM starts in the first state of that method's state-transition machine, executes the task associated with that state, and then selects a transition from that state. Each transition either terminates the method or leads to another state. A task associated with a state in a method can be either primitive or non-primitive (an unimplemented task cannot be executed; it must be adapted instead). If the task is non-primitive, a method is selected for it, etc. The execution of primitive tasks simply involves performing the action specified (e.g., executing an attached procedure). Throughout the execution process, a trace is built and errors are checked for. The particularly challenging portion of the execution of TMKL models is the decision making which takes place when a task has multiple methods available or a method has multiple transitions available. Section 5.1 presents an overview of the algorithms for executing tasks and methods. After that, Section 5.2 provides additional details regarding the decision making portion of the algorithm. Finally, Section 5.3 describes the sorts of failures that can arise during execution and the way that they are addressed.

5.1 Algorithms for Execution

Table 21 presents the algorithm for task execution in REM.[1] The first step of the algorithm involves building a trace element and having it refer to the task and the knowledge state at the start of execution. After that there are checks to see if the required inputs for the task are available and that the required condition for executing the task holds.[2]. Next, if the task is primitive, then it is possible to directly perform it. If the task is not primitive, a method must be selected for the task. The selection of methods is done using REM's general decision making technique (which is detailed in Section 5.2). After selection, there is a simple check to ensure that a method has been

[1]The algorithms for execution in SIRRINE are similar to the ones for REM, but are less sophisticated. In particular, the decision making and failure detection portions of SIRRINE's execution algorithms are considerably less powerful.

[2]The CHECK statements in the algorithms presented in this section perform failure monitoring. In this section, it is assumed that all checks are met (i.e., that no failures are detected). If any check is not met, the algorithm immediately terminates and annotates the trace accordingly. The details of these processes are described in Section 5.3.

Algorithm **execute-task**(main-task)	
Inputs:	main-task: A task to be executed
Outputs:	main-task-trace: A task-trace for the execution
Other Knowledge:	current-knowledge-state: The knowledge used and produced
Effects:	The task has been executed.

```
main-task-trace = NEW task-trace
main-task-trace:task = main-task
main-task-trace:input-knowledge-state = current-knowledge-state
CHECK [that the parameters in main-task:input are bound in current-knowledge-state]
CHECK [that main-task:given holds]
IF [main-task is primitive] THEN
   [do main-task]
ELSE
   chosen-method = DECIDE [on an applicable method for main-task]
   CHECK [that a method was found]
   chosen-method-trace = execute-method(chosen-method)
   main-task-trace:invoked-mmethod-trace = chosen-method-trace
[update bindings in current-knowledge-state]
main-task-trace:output-knowledge-state = current-knowledge-state
CHECK [that main-task:makes holds]
RETURN main-task-trace
```

Table 21: Algorithm for task execution

chosen. That method is then executed, and the trace generated for it is added to the trace for the main task. After the implementation of the task is completed (either by performing the primitive computation or by performing a method for the task), the bindings of the current knowledge state are updated, using the current values of the output parameters for the task (which may have been altered directly by the effects of a primitive or indirectly during the course of method execution). Next, the trace for the task is updated to refer to the updated knowledge state. Then there is a check to see if the desired effect of the task was accomplished. Finally, the trace of the task is returned.

Table 22 shows the algorithm for the execution of a method. First a method trace is built for the method. The remainder of the algorithm involves stepping through the transitions and reasoning states of the method. The first transition is specified by the method. A trace element is built for that transition and the method trace is linked to it. Then the loop begins. Each transition refers to exactly one reasoning state; a trace element is built for that reasoning state, and then the subtask for that trace is executed. Next, the reasoning state trace is made to refer to the execution trace for the subtask. Then a new transition is selected. This selection, like the selection of methods for a task, is done using the general decision making technique in Section 5.2. There is a check to ensure that a transition was selected, and a new trace element is built for that transition, and the reasoning state trace is linked to it. At this point, the process returns to the start of the loop unless a terminal transition has been reached; when it has, the method trace is returned.

5.2 Decision Making

Note that there are two separate decision making points in the algorithms in the proceeding section: deciding on a method for a task and deciding on a next transition within a method. Both of these decision making points invoke the same basic mechanism; in both cases the mechanism selects a single option from a collection of candidates (either methods or transitions). The decision making mechanism uses symbolic information encoded in the model to eliminate inapplicable candidates. If more than one candidate is applicable, it then uses reinforcement learning (specifically, Q-learning [Watkins and Dayan, 1992]) to select among alternatives.

Algorithm **execute-method**(chosen-method)	
Inputs:	chosen-method: A method to be executed
Outputs:	chosen-method-trace: A method-trace for the execution
Other Knowledge:	current-knowledge-state: The knowledge used and produced
Effects:	The method has been executed.

```
chosen-method-trace = NEW method-trace
chosen-method-trace:mmethod = chosen-method
current-transition = chosen-method:start
current-transition-trace = NEW transition-trace
current-transition-trace:transition = current-transition
chosen-method-trace:start-trace = current-transition-trace
WHILE [current-transition is not terminal]
  current-reasoning-state = current-transition:next
  current-reasoning-state-trace = NEW reasoning-state-trace
  current-transition-trace:next-reasoning-state-trace = current-reasoning-state-trace
  current-subtask-trace = execute-task(current-reasoning-state:subtask)
  current-reasoning-state-trace:subtask-trace = current-subtask-trace
  current-transition = DECIDE [on an applicable transition for current-reasoning-state]
  CHECK [that a transition was found]
  current-transition-trace = NEW transition-trace
  current-transition-trace:transition = current-transition
  current-reasoning-state-trace:next-transition-trace = current-transition-trace
RETURN chosen-method-trace
```

Table 22: Algorithm for method execution

5.2.1 Symbolic Decision Making for TMKL

Recall that both methods and transitions have provided slots which explicitly state the conditions under which they are applicable. The logical expressions in these slots provide the symbolic information which is used to filter out inappropriate decisions; each candidate has its provided slot checked and those candidates with expressions which are not true are rejected.

If there are any candidates which have no value in the provided slot at all, then those candidates are considered applicable if and only if no candidates which do have provided conditions are applicable. In other words, options which have no conditions attached to them are considered "last resort" defaults. This fact is a minor convenience but does not really affect the expressive power of TMKL; "last resort" defaults can always be alternatively expressed by taking the negation of the conjunction of all of the conditions for all of the other candidates. Note that TMKL does allow the specification of a candidate which should always be available, regardless of whether any other candidates are also available; such a candidate should simply have its provided slot contain the the Loom expression, :true.

Many tasks have exactly one method encoded for them, and many methods have exactly one transition leading out of each reasoning state. Furthermore, it is generally expected that if a task or reasoning state does have multiple candidates, that these candidates will typically have mutually exclusive provided conditions. For example, the Display Interpreted File task in the web browsing agent has one method which displays files internally and one which displays files externally. The provided conditions for these methods require that the MIME tag indicate an internal display type or an external display type, respectively. Since no tag can indicate both, these conditions always identify exactly one of these two methods to execute. The expectation that provided conditions be mutually exclusive is a reflection of the fact that a TMKL model is intended to encode a particular reasoning strategy; if the model does not specify what to do next at some point, then it hasn't completely described that strategy. REM does not *require* that models be complete in this way, but models should at least be nearly complete (i.e., have very few of these ambiguous decision points) for REM to be able to perform efficiently and effectively.

Thus the filtering of candidates using the provided slots should identify exactly one candidate in most circumstances. When exactly one candidate is found, it is simply returned immediately by the decision making process.

However, when multiple candidates are still available at this point, REM needs to select among them. If REM has had these same options available at the current point in the model earlier in the execution process, it simply reuses the decision that it made the last time. Otherwise, it uses reinforcement learning, specifically Q-learning.

5.2.2 Reinforcement Learning for TMKL

The mechanism for reinforcement learning is a relatively straightforward application of Q-learning. The details of Q-learning have been thoroughly explained elsewhere, e.g., [Watkins and Dayan, 1992, Kaelbling et al., 1996, Mitchell, 1997]; however, below is a brief overview of the characteristics of Q-learning which are particularly relevant to its use in REM:

1. The inputs for a particular decision to be made by Q-learning include a representation of the current state and a set of alternatives to select among.
2. The central assumption of Q-learning is that the Markov property holds, i.e., that the likelihood of each outcome for a given alternative in a given state depends only on the state and not at all on the past history. If the Markov property does not hold then the Q-learning algorithm may fail to converge or may converge to a non-ideal policy.
3. The Q-learning process maintains a record of a function, `Q`, with two arguments, `s` (a state) and `a` (an alternative or "action" in typical Q-learning terminology), and a numerical output. The value `Q(s,a)` is an estimation of the expected benefit from selecting `a` in `s`.
4. When a decision needs to be made, alternatives with high `Q` values are preferred.
5. After a decision is made, the value of `Q` for that decision is updated based on both the immediate reward received and the greatest value of `Q` for all actions in the next state (i.e., alternatives which tend to lead to states which are expected to bring great rewards should presumably be valued highly).

The five characteristics of Q-learning above are reflected in REM's use of Q-learning. The relation of each of these points, respectively, has to REM are detailed in the five points below:

1. REM uses the different candidates (methods or transitions) as the alternatives for Q-learning to select among. The states which REM gives Q-learning consist of the task or reasoning state from which a method or transition is being sought *plus* a complete history of all decisions made by Q-learning in the past.
2. Even with a complete history of decisions, the states used for Q-learning do not completely conform to the Markov property. There are aspects of the condition of the system which are not reflected in this formalization of state.
3. REM represents the `Q` function using a hash table mapping pairs of states and alternatives to values. If a `Q` value is requested for which there is no entry in the table, that value is assumed to be 0 (i.e., all `Q` values are implicitly initialized to 0).
4. REM has two different mechanisms for selecting an option given the set of `Q` values: selection of maximum `Q` values and Boltzmann Exploration (as described in [Kaelbling et al., 1996]). REM can be configured by the user to use either mechanism.
5. REM gives a fixed reward at the end of execution if that execution is successful (i.e., the makes condition for the main task is satisfied). It does not provide any positive or negative reinforcement at any other point during execution.

Note that a number of the features described above are not necessarily ideal. In particular, the representation of states described in items 1 and 2 seems potentially very costly; while encoding a complete history of decisions in the state seems to work in the examples that have been tried, it does lead to a state space which is exponential with respect to the number of decisions made during a reasoning episode (and it still does not completely address the Markov property issue). With respect to item 2, it is important to note that the use of the complete history of decisions does come significantly closer to conforming to the Markov property than just using the current decision

point (i.e., the current task or reasoning state). The results presented in Chapter 7 suggest that the approach used by REM is close enough to conformance to be effective on the sorts of problems we have considered so far without being overwhelmingly costly (at least when the model localizes the reinforcement learning to a few key decisions).

Item 3 also raises some interesting issues: how should `Q` values be represented and initialized. In particular, it seems plausible that, under some circumstances, qualitative information in the model should be able to provide some insights into which options are likely to productive under different circumstances. These insights could potentially be used to provide better initial `Q` values than simply starting with 0. This seems like an important topic for future research.

The two different selection mechanisms described in item 4 provide substantially different behavior. One of them is very straightforward: always selecting the highest `Q` value. This can be very efficient because once it finds an acceptable solution, it sticks with it as long as it remains acceptable. It can even be a flexible solution; if the environment changes in such a way that the the currently acceptable solution becomes unacceptable, its `Q` value will go down until another solution is preferred. However, it is very easy for such an approach to miss opportunities in a non-deterministic and/or dynamic environment by over-relying on past results. Furthermore, even in static environments this approach is extremely susceptible to being stuck in locally optimal solutions. These problems are addressed by REM's other selection mechanism, Boltzmann Exploration. That approach generally favors alternatives with high `Q` values, but also adds a limited amount of randomness to ensure that a variety of alternatives are attempted. However, Boltzmann Exploration can be unnecessarily inefficient in static, deterministic domains which lack local optima (because the mechanism can chose options which are already known to be ineffective). Consequently, REM defaults to maximum `Q` value selection and requires that a user explicitly switch to Boltzmann Exploration when it is needed. In the experiments described in Section 7.1, only the processes involving the situated learning mechanism use Boltzmann Exploration; exploration is typically essential for processes produced by that mechanism because those processes have no specific strategy encoded in them. One possible topic for future work is a more detailed analysis of when different selection mechanisms should be used; such an analysis may even allow autonomously selecting the appropriate mechanism for a given domain.

Item 5 also presents some theoretically interesting issues. The use of a single, fixed reward at the end of execution ensures that paths through the state space that lead to success will be favored over those that do not. Furthermore, because Q-learning discounts rewards over multiple actions, this approach will favor shorter paths over longer ones (since the latter have the rewards more heavily discounted). Thus this reward structure does favor a reasonably good ultimate outcome: i.e., that alternatives which lead to success are always favored over ones which don't, and among alternatives which lead to success, those which lead more directly to success are favored over those that do so more indirectly. There do seem to be some drawbacks to this approach. There is no indication of the degree of success (because TMKL only encodes conditions for success, not quantitative metrics). Adding this information would allow Q-learning to improve not only the likelihood of success but also the quality of results. Furthermore, there is no notion of intermediate success (i.e., rewards received during execution instead of at the end). Adding this information would allow Q-learning to learn from fewer executions because it would encourage execution paths which accumulate these intermediate rewards (assuming that they are more likely to lead to success). The use of a more elaborate reward structure and the ways in which reward structure information can be integrated with functional models seems like yet another productive topic for future research.

Ultimately, it is not clear whether the decision making component of REM would be more effective if it used different alternatives for some of the options above, or a different reinforcement learning mechanism, or another approach altogether (e.g., search). These issues are addressed in more detail in Section 9.1.3.

5.3 Failures

The discussion of the execution algorithms in Section 5.1 focused on the behavior of these algorithms in the context of ordinary, successful computation. REM is also capable of reacting to exceptional circumstances, i.e., ones in which the system being executed fails in some way. In particular, the algorithms presented in Section 5.1 have a total of five CHECK statements. For each such statement, if the check is met, the algorithm proceeds normally. If, however, the check condition is not met, the following events occur:

1. A failure annotation is generated; each of the five checks produces exactly one type of failure annotation.

2. The failure annotation is added to the failures slot of the current trace element.

3. Any additional information which belongs in the failure annotation is added to that failure.

4. The execution of the current task or method stops.

After a failure is detected, no more tasks or methods may be invoked. However, the execution processes currently running are allowed to finish. In particular, the construction of traces and output knowledge states for the higher level tasks and methods is completed; REM will still build the trace elements and the failure annotations for high-level tasks and methods when a low-level task or method has failed during the execution. Thus it may often be the case that a particular incident during execution produces a number of failure annotations at different levels in the hierarchy. For example, if no available transition is found for a reasoning state while executing a method, then a failure annotation is first attached to the trace element for that reasoning state. Next, if the task which invoked that method has a makes condition which is not met (which is likely since the method for the task failed), then a failure annotation is also attached to the trace element for that task. Furthermore, if that task was invoked within a method for a higher-level task, then that higher level task may also have a makes condition which is not met, creating yet another failure annotation.

The primary purpose of the failure annotations is to support REM's failure-driven adaptation process (described in Section 6.4). The fact that a failure is attached to a particular element in a trace helps to localize reasoning about that failure; however, since a particular incident may generate multiple failure annotations, additional reasoning is required to further narrow the localization. In addition, the type of the failure may suggest some mechanism for repair. For example, if the failure involves having no applicable methods, then repair may involve constructing a method which is applicable in that situation. Lastly, the information in the failure annotation and the trace element to which it is attached may be needed to accomplish the mechanism for repair. For example, constructing a method which is applicable in some situation may require knowing the knowledge state which was present in that situation; that information occurs in the input-knowledge-state slot of the task trace.

The following outline describes the five different kinds of failures that REM can detect during execution. For each failure type, the outline lists the algorithm in which it is produced (from Section 5.1), the CHECK statement that causes it to be produced (copied from the algorithms), and the additional information, if any, which is encoded in the failure annotation. The additional information appears in the form of slots in that failure annotation type; note, however, that most of the failure annotation types below do not require additional information because all of the important information about the failure is encoded in the trace element to which the annotation is attached.

- missing-inputs

  Algorithm : execute-task

  Statement : CHECK [that the parameters in main-task:input are bound in current-knowledge-state]

  Additional Information : (inputs: a list of parameters in main-task:input which are not bound in current-knowledge-state)

- given-fails

  Algorithm : execute-task

  Statement : CHECK [that main-task:given holds]

  Additional Information : None

- no-applicable-mmethod

  Algorithm : execute-task

  Statement : CHECK [that a method was found]

  Additional Information : None

- makes-fails

  Algorithm : execute-task

  Statement : CHECK [that main-task:makes holds]

  Additional Information : None

- no-applicable-transition

  Algorithm : execute-method

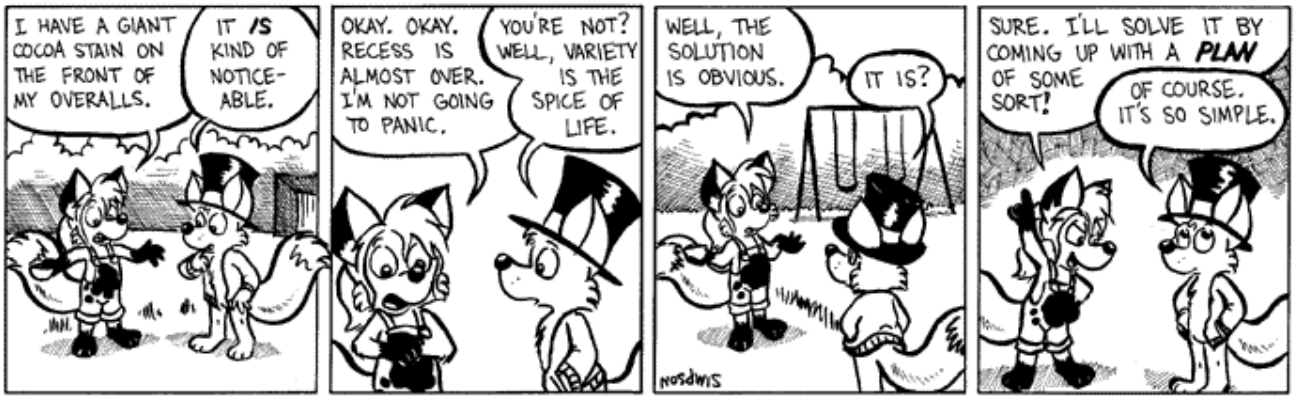
  Statement : CHECK [that a transition was found]

  Additional Information : None

Note that this set of five types of failures follows fairly naturally from the nature of TMKL and it's execution; these are the five apparent ways in which a ontologically valid TMKL model can fail. Of course, there are vastly more ways in which an invalid TMKL model can fail; REM does not address failures of this sort. For example, there is a failure type, no-applicable-transition, which indicates that there is no transition which can be selected in a given reasoning state. One could also envision a failure in which there is no subtask for a given reasoning state. However, the definition of TMKL (in Chapter 2) explicitly requires that each reasoning state have exactly one subtask and (consequently) provides no conditions on the selection of a subtask for a reasoning state. Thus the only possible way for there to be no subtask for a given reasoning state is for the reasoning state to be invalid to begin with. In contrast, a reasoning state may have many transitions and each transition may have a provided condition which must be met to select it. A TMKL model may be ontologically valid, and may, indeed, even be able to operate correctly in many situations, but could still encounter an obscure situation in which there is no transition with a provided condition that is met. The reason that REM does not detect or address inherent flaws in a model is that the focus of REM is on adaptation of agents which work for one class of problems into agents which work for another (typically broader) class of problems; REM is not intended to perform adaptation on agents which don't yet work at all.

Chapter 6

Adaptation



"Ozy & Millie" © 1999 D. C. Simpson

When an agent is not able to do what the user wants done, adaptation needs to be performed. Adaptation results in a modified model which can then be executed using the mechanisms described in Chapter 5. There are four general varieties of adaptation which have been performed in this research: situated learning, generative planning, proactive model transfer, and failure-driven model transfer. The first two of these varieties require relatively simple knowledge: the primitive actions available, the task to be performed, and the specific inputs to that task. These approaches create a new method for the given task using those actions. The other two varieties involve transfer from an existing model. Such a model includes any primitive actions performed by that agent and also includes one or more existing methods for performing the tasks of that agent. The model transfer techniques involve modifying the existing methods (including the subtasks and lower level methods) to suit the new demand. The combination of these four approaches provides a powerful basis for adaptation in a variety of circumstances.

Recall from Chapter 1 that adaptation is performed in two different circumstances: when a new task which has no method is requested by a user and when the execution of a task has not produced the desired result. In either case, an adaptation strategy needs to be selected. In SIRRINE, there is a very limited range of adaptation strategies (only some minor variants of the failure-driven model transfer process described in Section 6.4); selection of strategies in SIRRINE is performed using simple hard-coded applicability rules. In contrast, REM has a broader range of adaptation strategies and a more principled account of strategy selection. In REM the overall process of adaptation and execution is explicitly modeled in TMKL; i.e., REM has a TMKL model of itself. There is a general Adapt task which appears in the model wherever adaptation is needed. There are four different methods for the Adapt task which involve the four different kinds of adaptation in REM (situated learning, generative planning, proactive model transfer, and failure-driven model transfer).

Because the alternative adaptation strategies are methods for the same task, selection among them is done using the decision making mechanism described in Section 5.2. Recall that this mechanism involves using logical conditions on methods to isolate viable candidates and then uses reinforcement learning to select among these candidates. The logical conditions on the adaptation strategies in REM were deliberately built to be mutually exclusive so that only

one candidate is ever viable and no reinforcement learning needs to be done for selection of adaptation strategies.[1] The first two strategies (situated learning and generative planning) can be attempted for any adaptation process because they do not depend at all on characteristics of an existing model. Proactive model transfer can be attempted whenever there is an existing task which is similar to the desired task and has a method. Failure-driven model transfer can be attempted whenever there is an existing trace for the main task in the current situation (i.e., the problem has already been attempted). The logical conditions that REM uses to select adaptation methods encode those facts and also include some *ad hoc* terms to avoid having conflicting strategies available (e.g., the generative planning method condition includes a requirement that there is no similar task for which proactive model-transfer could be used).

Figure 23[2] presents the situated learning mechanism. This process is very simple: it just creates a new method which allows all of the actions which the agent can perform at each step of the process. The combination of the given task and the new method for it constitute a model which can then be executed. Because the model allows all possible actions at each step, all action selection is done during execution using the reinforcement learning mechanism in Section 5.2.2.

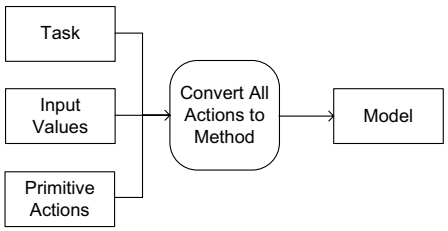


Figure 23: An overview of the situated learning adaptation process.

The overall mechanism for adaptation by generative planning is depicted in Figure 24. The process involves first converting the available information about operators and facts into a form which can be used by a traditional planning system. Next, that planning system is invoked on those operators and facts in an attempt to satisfy the requirements of the given task. Finally the plan generated is generalized into a method for that task. Since the task now has a method, there is now a complete model which can be executed.

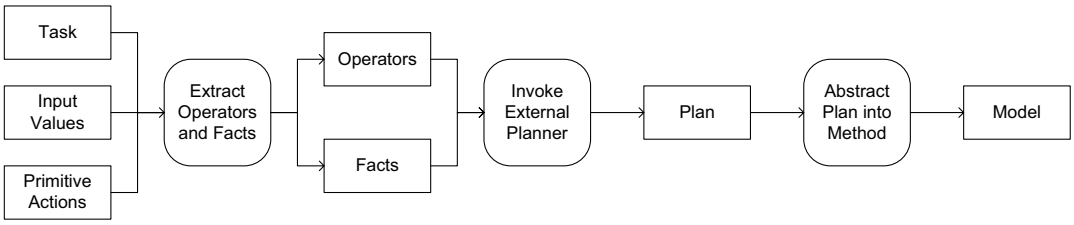


Figure 24: An overview of the generative planning adaptation process.

Figure 25 describes the overall process for proactive model transfer. First a task from the existing model which has a method and is similar to the new task is retrieved. Next some sort of modification is performed to adjust the method (plus subtasks, etc.) for the retrieved task to suit the new task. In principle, there are any number of techniques which could be used to accomplish this modification. The one process which has been developed in this work for such modification is called relation mapping. The end result of proactive model transfer is a modification to the model (specifically, the given task now has a method in the model).

The final adaptation mechanism, failure-driven model transfer, requires that there be an existing trace for the model given a set of inputs. Consequently, it is only appropriate when the task being performed is already in the

[1]Recall from Section 5.2 that this trait is considered desirable when possible; an agent is likely to be much faster if it has been told exactly what to do in each possible situation. On the other hand, it can also be limiting in that it prohibits the use of otherwise applicable methods which could actually be preferable in some situations. One option might be encoding preferences for strategies instead of requirements, i.e., using logical assertions to bias reinforcement learning instead of circumventing it. This would allow strategies which are usually best in some situation to be selected more often but not always. REM's decision making mechanism does not currently allow that; for more information on limitations and potential future work on the decision making component, see Section 9.1.3.

[2]In this and the other figures which appear at the start of this chapter, rounded boxes represent processes and rectangular boxes represent knowledge items. Arrows from knowledge items to processes indicate inputs. Arrows from processes to knowledge items indicate outputs.

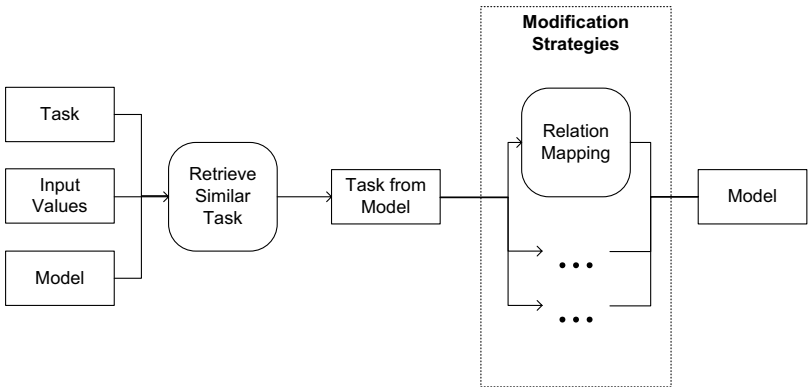


Figure 25: An overview of the proactive model transfer adaptation process.

model and has a method; traces are produced during execution so there must have been something to execute for there to be a trace. The adaptation mechanism addresses differences between what did actually happen during execution and what should have happened. The first step in the process involves analyzing feedback provided by the user (if any). This step can place annotations on a trace indicating where there are contradictions between what should have happened and what did happen. These annotations are in addition to any annotations which were placed on the trace during execution (as described in Section 5.3). After all the failure annotations are available, they are considered one at a time to determine whether one of the existing repair processes is able to address the failure. The two repair processes used in this research are fixed value production and generative planning; the latter involves the same reasoning process which is used for general adaptation by generative planning, but does so for a specific subtask which has been labeled with an appropriate failure mechanism. After repair is done by one of these two mechanisms, the model is revised and can be executed again.

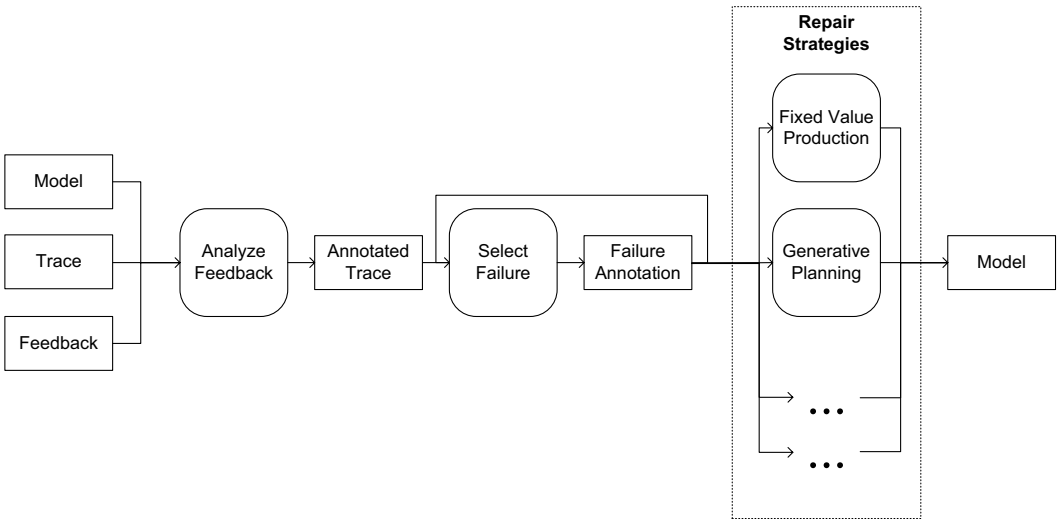


Figure 26: An overview of the failure-driven model transfer adaptation process.

These four types of adaptation together provide an overall mechanism for altering models to suit the demands of new tasks or new situations for existing tasks. They are described in four separate sections below.

6.1 Situated Learning Adaptation

Given only a description of a task, some inputs, and a description of available actions, the options for addressing that task are fairly limited. One approach that can be taken in this situation is to address the problem by pure Q-learning, i.e., to simply try arbitrary actions and see what happens. The situated learning strategy in REM takes

this approach. The term "situated" is intended to indicate that behavior is guided entirely by action and experience, and does not involve explicit planning or deliberation. The situated learning mechanism in REM creates a method for the given task which contains no advance knowledge about how to address that task; the method makes the widest variety of actions possible at any given time and then relies on the Q-learning portion of model execution to select among available actions.

Figure 27 provides a partial sketch of the tasks and methods generated by this approach for the ADDAM assembly example on the roof example (from Section 4.1.1) with only two boards. The description of the Assemble Roof-2 task is given to REM; this description includes a description of the required state at the end of the task (i.e., that the two boards are in place and are screwed together). The description of the actions, such as screw and place, are also given to REM; these descriptions include not only expressions which must hold before and after the actions (e.g., after the screw action, the objects are connected and are not free to be moved) but also links to code which simulates the effect. The remaining elements of Figure 27 are created by REM.

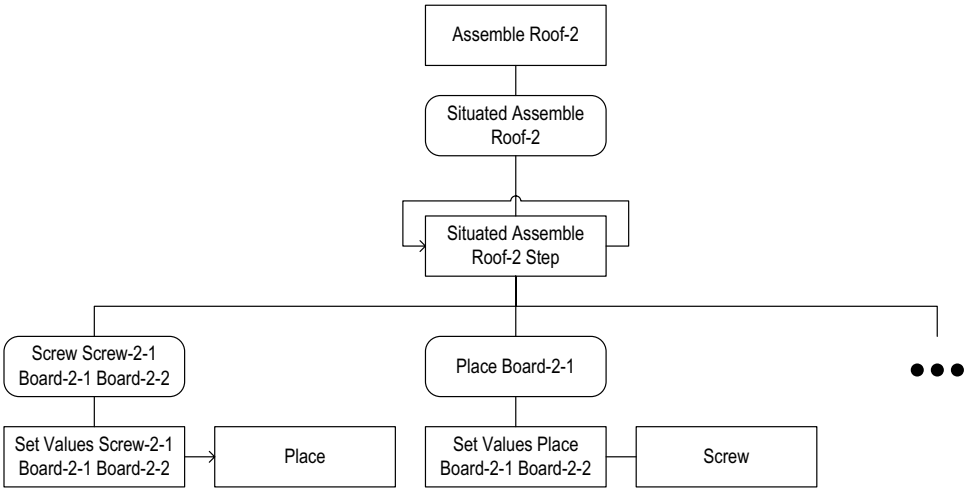


Figure 27: A diagram of the tasks and methods involved in the assembly process produced by situated learning adaptation for the two-board roof example. The ellipses at the end indicate that there are many more methods for the step task, each having a form similar to the two presented.

The overall method generated for this problem is called Situated Assemble Roof-2. This method is very simple; it just executes the Situated Assemble Roof-2 Step task repeatedly until the required state for the overall task is met or no more actions are possible. The step task has many methods: one for each possible combination of an action and a set of objects. For example, the first method listed involves screwing a specific screw into two specific boards. This method consists of first setting the parameters for the screw action to be those values and then invoking the screw action on those parameters. The provided condition for each method is the given condition the task, instantiated for the particular values used in that method. For example, the given condition for the screw task requires that the screw be unblocked and not in use, and that both boards be in position. The provided condition for the first method in the figure thus requires that Screw-2-1 be unblocked and not in use, and that Board-2-1 and Board-2-2 be in position. Thus at each step, REM will only be able to select among all method for which the specified action is currently possible for the specified values.

Table 23 presents the algorithm for situated learning. First, a new main method is created and added to the main task. Next, a new subtask is created for the new method; the method specifies that the subtask is executed repeatedly until the makes condition for the main task is met. After that, there is a loop through all known actions. The first few steps of the loop involve creating a list of all possible combinations of input parameters and values. Next, there is a sub-loop which steps through each of these input-value combinations. Within the sub-loop a new method, new-action-method, is created and added to the subtask of the main method. The provided condition for new-action-method is created by converting any references to input parameter values in the given condition of the action to values in the input-value combination. For example, the given condition for the unscrew action in ADDAM is:

```
(:and
```

```
 (free (value current-object))
 (screws (value current-object) (value current-subobject-1))
 (screws (value current-object) (value current-subobject-2)))
```

If this task is being used in situated learning and the particular input-value combination being processed is (Screw-2-1 Board-2-1 Board-2-2), then the provided clause for new-action-method is:

```
(:and
 (free Screw-2-1)
 (screws Screw-2-1 Board-2-1)
 (screws Screw-2-1 Board-2-2)
```

After the provided condition has been built, a new task is created. The subtasks for new-action-method are set to be this new task followed by the primitive action. Finally, the new task is made to bind the input parameters for the primitive action to the values specified in the input-value combination, so that whenever new-action-method is invoked, the primitive action will be performed on those values for those inputs. After all of the loops have completed, the main task, which now has a method, is returned.[3]

6.2 Generative Planning Adaptation

Generative planning is another approach which can be taken using only information about the desired task, inputs, and available actions. The process for using generative planning in REM involves sending the relevant information to an external planner and then turning the plan that it returns into a method for the desired task. REM uses Graphplan[4] [Blum and Furst, 1997] as its external planner. The generative planning mechanism in REM can be used not only before execution to construct a method for a novel main task, but also after a failure to construct a method for a faulty subtask. The example presented in this section focuses on the former use. Section 6.4 includes an example in which the same generative planning mechanism is localized to particular subtask.

Figure 28 shows the tasks and methods generated through generative planning for the ADDAM assembly example with the two board roof. The roof with only two boards is very simple and consequently the plan produced is very simple: each board is placed and then the two of them are screwed together. The transitions within the method produced control the flow of parameter values (e.g., ensuring that the first board is the argument to the first call to Place).

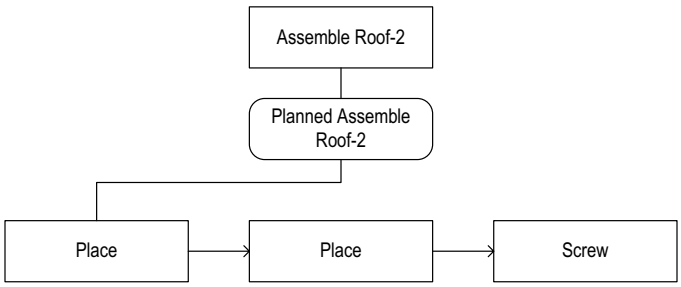


Figure 28: A diagram of the tasks and methods involved in the assembly process produced by generative planning for the two-board roof example.

[3]Incidentally, note that Figure 23 at the start of this chapter depicted the output of this process as a model, not a task. However, the task which is returned is linked to a method, and those methods are linked to subtasks, and the various tasks and methods are linked to parameters, and the parameters are linked to concepts, etc. Consequently, the algorithm-level result of returning a task with a method is the equivalent of the conceptual-level result of returning a model. This equivalence is also present (implicitly) in the diagrams of the other algorithms in this chapter.

[4]The code for Graphplan was downloaded from http://www.cs.cmu.edu/afs/cs.cmu.edu/usr/avrim/Planning/Graphplan/. The code is Copyright 1995 by Avrim Blum and Merrick Furst. The code includes explicit permission for non-commercial research use (as is done in REM). The Graphplan executable used in the work described in this dissertation was compiled with the following constant values: MAXMAXNODES was set to 32768 and NUMINTS was set to 1024. Other than setting these values, the original source code has not been modified.

Algorithm **adapt-using-situated-learning**(main-task)	
Inputs:	main-task: The task to be adapted
Outputs:	main-task: The same task with a method added
Other Knowledge:	current-knowledge-state: The knowledge to be used by the task
	primitive-actions: A set of primitive actions which can be performed
Effects:	A new method has been created which accomplishes main-task within current-knowledge-state using the primitive-actions.

```
new-main-method = NEW method
[add new-main-method to main-task:by-mmethod]
new-step-task = NEW task
[subtask for new-main-method] = new-step-task
[control for new-main-method] = [repeat until main-task:makes holds]
FOR primitive-action IN primitive-actions DO
      input-value-lists = NEW list
      FOR input-parameter IN primitive-action:input DO
            [add list of all known instances of input-value:concept input-value-lists]
      input-value-combinations = [all possible combinations of values in input-value-lists]
      FOR input-value-combination IN input-value-combinations DO
            new-action-method = NEW method
            [add new-action-method to new-step-task:by-mmethod]
            new-action-method:provided = [substitute input-value-combination for
                                          parameter references in primitive-action:given]
            new-set-task = NEW task
            [subtasks for new-action-method] = new-set-task, primitive-action
            [control for new-action-method] = [do once in series]
            new-set-task:binds = [bind primitive-action:input parameters to
                                  input-value-combination]
RETURN main-task
```

Table 23: Algorithm for adaptation by situated learning

Algorithm **adapt-using-generative-planning**(main-task)	
Inputs:	main-task: The task to be adapted
Outputs:	main-task: The same task with a method added
Other Knowledge:	current-knowledge-state: The knowledge to be used by the task
	primitive-actions: A set of primitive actions which can be performed
Effects:	A new method has been created which accomplishes main-task within current-knowledge-state using the primitive-actions.

operators = [translate primitive-actions]
relevant-relations = [all relations in any slot of main-task or any of primitive-actions]
assertions = [translate all facts in current-knowledge-state involving any relevant-relations]
goal = [translate main-task:makes]
objects = [names of all objects which are directly referenced in facts or goal]
object-types = [the concepts for which the objects are instances]

plan = [invoke planner on operators, assertions, objects, object-types, and goal]

new-method = NEW method
[subtasks for new-method] = [extract actions from plan]
[transitions for new-method] = [infer transitions from plan]
[add new-method to main-task:by-mmethod]
RETURN main-task

Table 24: Algorithm for adaptation by generative planning

Table 24 presents the algorithm for the REM adaptation strategy which uses generative planning. The first few steps involve taking information in REM and translating it into a form which Graphplan can manipulate (facts and operators). The computation of the operators is fairly straightforward; those primitive tasks which involve action are simply represented in a form which can be processed by Graphplan. If a primitive task involves a procedure, that procedure is omitted from the operator; Graphplan is only given the stated logical results of the execution of the task, not the task's implementation. Note that not all TMKL actions may be able to be translated in to Graphplan's operator language. The results of a task are described in REM as an arbitrary Loom logical expression; such expressions can contain a wide variety of constructs which are not supported by Graphplan (e.g., disjunction, quantification, etc.). Furthermore, primitive tasks with procedures are not required to have their results stated at all; primitive tasks whose results are not described or cannot be translated into Graphplan's planning language are omitted from the set of operators. The inability to translate all possible TMKL actions is an inevitable limitation of the planning mechanism: the planner can only reason about those actions which it can represent. The use of a different planning algorithm with a different language for representing operators could allow some operators which cannot be represented in Graphplan. However, it is certainly true that not all problems which can be described in TMKL can be successfully solved by the generative planning mechanism; the fact that not all TMKL actions can be translated into planning operators is one reason for this fact.

After the operators are translated, the assertions which constitute the starting condition for the planner are extracted. Obtaining these assertions involves first finding the various relations which are relevant to the main task and the actions, i.e., those referred to in the given, makes, and asserts slots of the task (which are the three slots of a task which can contain arbitrary combinations of concepts, relations, and logical operators). Next, the process finds all connections between values known at the start of execution which involve those relevant relations. These assertions are also translated into a form which Graphplan can process. Next the goal (which appears in the makes slot of the main task) is translated. Finally, the objects and types in the domain are translated as well. Graphplan has a rudimentary mechanism for indicating types of objects, but it doesn't allow even simple type representation capabilities such as subtyping. Consequently REM translates object types into logical assertions when exporting to Graphplan rather than using Graphplan's typing mechanism. These assertions are merged with the ones extracted from the relations in REM and the combination is represented in Graphplan as the problem preconditions.

```
(operator EXECUTE-PLACE
  (params (<current-object>))
  (preconds (physical-element <current-object>)
   (physical-component <current-object>) (free <current-object>))
  (effect (present <current-object>)))

(operator EXECUTE-SCREW
  (params (<current-object>) (<current-subobject-1>)
   (<current-subobject-2>))
  (preconds (physical-element <current-object>)
   (physical-element <current-subobject-1>)
   (physical-element <current-subobject-2>)
   (physical-connection <current-object>)
   (physical-component <current-subobject-1>)
   (physical-component <current-subobject-2>) (free <current-object>)
   (loose <current-object>) (present <current-subobject-1>)
   (present <current-subobject-2>))
  (effect (screws <current-object> <current-subobject-1>)
   (screws <current-object> <current-subobject-2>)
   (del loose <current-object>) (del free <current-subobject-1>)
   (del free <current-subobject-2>)))
```

Table 25: Two of the operators provided to Graphplan for the the assembly and disassembly examples.

After all of the translation is done, the results are sent to two separate files which Graphplan requires as inputs: a "facts" file which contains the objects, assertions, and goal plus an "operators" file which contains the operators. Table 25 shows a couple of the operators in the ADDAM domain which are produced by REM for use in Graphplan. Each operator is divided into three sections: parameters, preconditions, and postconditions. For example, consider the EXECUTE-PLACE operator. It has one parameter: the object to place. It also has three preconditions: two indicate the type of the object (it is a physical-element, and it is also a physical-component, which is a subclass of physical-element within the TMKL model) and one indicates the fact that the component must be free to move about. Lastly, the action has one effect: i.e., that it is now present in the device.

Table 26 shows the entire "facts" file for the two-board roof example. The first three lines declare the existence of the objects in the domain. The remaining sections describe the state of the world before the plan and the desired state at the end, respectively. With only two boards, these states are very simple, particularly since there are no obstructions in the two-board design (there aren't enough components for anything to get in the way of anything else). With more boards, the problem becomes both larger and more intricate.

After the files are produced, Graphplan is executed.[5] Table 27 shows the plan returned by Graphplan for the two-board roof. Again, with only two boards, the design is very simple; the resulting plan is correspondingly simple: both boards are put into place and then they are screwed together.

The next few steps of the adaptation through planning algorithm involve converting the plan into a method. First, a new method is produced. The subtasks of that method are simply the actions in the plan. A particularly challenging aspect of converting a plan into a TMKL method involves inferring transitions between the subtasks. Recall (from Chapter 2) that transitions are used both to guide the ordering of subtasks within a method and to indicate the links between the parameters of the subtasks. Building a method from a plan involves generating both of these types of information.

In principle, the building the ordering can be very complex; for example, if a plan consists of two actions repeated several times, it might be reasonable to infer that the general process should contain a loop. It is not always clear, however, when to infer that a particular trend in ordering really does represent a significant pattern which should

[5]In REM, Graphplan is executed by spawning an external process which runs a C shell script which first runs Graphplan and then pipes the output through a Tcl script which parses that output into a LISP-like form. The output in that form is then returned to the main REM process within LISP.

```
(left-board-2-1-instance-1)
(right-board-2-1-instance-1)
(roof-screw-2-1-instance-1)
(preconds (physical-component left-board-2-1-instance-1)
 (free left-board-2-1-instance-1)
 (physical-element left-board-2-1-instance-1)
 (physical-component right-board-2-1-instance-1)
 (free right-board-2-1-instance-1)
 (physical-element right-board-2-1-instance-1)
 (loose roof-screw-2-1-instance-1)
 (physical-connection roof-screw-2-1-instance-1)
 (free roof-screw-2-1-instance-1)
 (physical-element roof-screw-2-1-instance-1))
(effects (present left-board-2-1-instance-1)
 (present right-board-2-1-instance-1)
 (screws roof-screw-2-1-instance-1 left-board-2-1-instance-1)
 (screws roof-screw-2-1-instance-1 right-board-2-1-instance-1)
 (free roof-screw-2-1-instance-1))
```

Table 26: The problem file provided to Graphplan for the two-board roof example.

```
EXECUTE-PLACE RIGHT-BOARD-2-1-INSTANCE-1

EXECUTE-PLACE LEFT-BOARD-2-1-INSTANCE-1

EXECUTE-SCREW ROOF-SCREW-2-1-INSTANCE-1 RIGHT-BOARD-2-1-INSTANCE-1
              LEFT-BOARD-2-1-INSTANCE-1
```

Table 27: The plan generated by Graphplan for the two-board roof example.

be represented in the method. The adaptation by planning mechanism in REM uses a very simple solution to the ordering problem: it puts the steps of the method in the same order as the steps of the plan with no branches or loops. It may be productive for future work on this topic to consider a broader range of alternatives for ordering steps in a method which was constructed from a plan.

REM's mechanism for constructing links between the parameters of subtasks is somewhat more sophisticated. That mechanism begins by assuming that operations in the plan which refer to common objects should access each of those objects by linking those objects' parameters within the method. For example, it is assumed that the appearance of RIGHT-BOARD-2-1-INSTANCE-1 in the first step of the plan in Table 27 and the appearance of the same value in the third step of the plan are related and thus that there should be a link between the current-object parameter in the first subtask of the method and the current-subobject-1 parameter in the third subtask. Note that the process of inferring links for parameters by first assigning parameters to the input values and then propagating those bindings through the plan is essentially a form of Explanation-Based Generalization [Mitchell et al., 1986]. The method produced is certainly a generalization of the plan; it represents an abstract description of how the process can occur, for which the plan is a specific instance. The links between parameters can be viewed as essentially a form of explanation in that they bridge the gap between the values and the parameters and thus explain how the effects of the actions in the plan, which are expressed as specific values, accomplish the overall goal of the main task, which is typically expressed in terms of the task's parameters.

Once the transitions for the method are completely specified, the method is ready to be used. Next, the specification of the main task is revised to indicate that the newly constructed method is one which can accomplish that task. At that point, the adaptation is done. When the resulting task is executed, the new method can be selected, and the steps of the plan can be performed.

6.3 Proactive Model Transfer

REM has a substantial advantage when it is given not only a task and a set of actions but also a method for accomplishing a similar task. For example, if REM is given not only a description of assembly and some assembly actions but also a model of a process for disassembly, it can try to transfer its knowledge of disassembly to the problem of assembly. Figure 29 shows some of the tasks and methods of the ADDAM hierarchical case-based disassembly agent which are particularly relevant to this example. For a more detailed description of the ADDAM model, refer to Section 4.1.

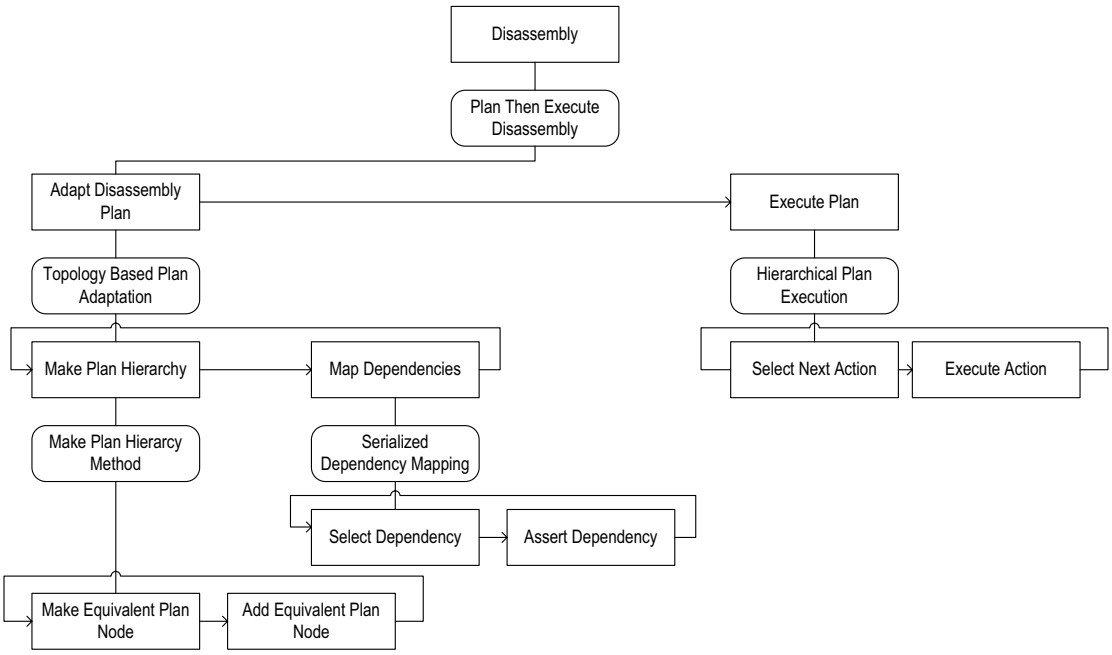


Figure 29: A diagram of some of the tasks and methods of ADDAM.

Algorithm **proactive-model-transfer**(main-task)	
Inputs:	main-task: The task to be adapted
Outputs:	main-task: The original task with a method added
Other Knowledge:	current-knowledge-state: The knowledge to be used by the task
Other Knowledge:	known-model: A pre-existing TMKL model
Effects:	A new method has been created which accomplishes main-task within current-knowledge-state using a modified version a method in known-model.

known-task = [retrieve T in known-model such that (similar-to main-task T) holds]
main-task = relation-mapping(main-task, known-task)
RETURN main-task

Table 28: Algorithm for proactive transfer

Recall that the top level task of ADDAM is Disassemble. This task is implemented by ADDAM's process of planning and then executing disassembly. Planning in ADDAM involves taking an existing plan for disassembling a similar device and adapting it into a plan for the new device. ADDAM's planning is divided into two major portions: Make Plan Hierarchy and Map Dependencies. Make Plan Hierarchy involves constructing plan steps, e.g., screw Screw-2-1 into Board-2-1 and Board-2-2. The heart of the plan hierarchy generation process is the creation of a node for the hierarchical plan structure and the addition of that node into that structure. Map Dependencies involves imposing ordering dependencies on steps in the plan, e.g., the two boards must be put into position before they can be screwed together. The heart of the dependency mapping process is the selection of a potential ordering dependency and the assertion of that dependency for that plan. Note that dependencies are much simpler than plan nodes; a dependency is just a binary relation while a node involves an action type, some objects, and information about its position in the hierarchy. The relative simplicity of dependencies is reflected in the implementation of the primitive task which asserts them; this task is implemented by a simple logical assertion (in the task's asserts slot) which says that the given dependency holds. In contrast, the task which adds a plan node to a plan is implemented by an entire complex procedure. Given the collection of plan steps and ordering dependencies which the ADDAM planning process produces, the ADDAM execution process is able to perform these actions in a simulated physical environment. Execution involves repeatedly selecting an action from the plan (obeying the ordering dependencies) and then performing that action.

When REM is provided with ADDAM and a specification of assembly and is asked to assemble a device, it needs to perform some adaptation in order to have a method for this task. In this situation, any of three techniques could be used. Because the model for ADDAM includes its primitive actions, the two kinds of adaptation which combine primitive actions (situated learning and generative planning) could be applied. Alternatively, because REM has a model in this situation and because there is a task in this model which is similar to assembly, it is also possible to transfer information from the existing model to the new task. This model transfer process is referred to as "proactive" because it takes place prior to any sort of execution. In this situation, REM selects proactive transfer over the techniques which do not use models (as described at the start of this chapter).

Table 28 shows the general proactive transfer algorithm used in REM. The first step in the process involves retrieving an existing task which is similar to the task to be adapted. In REM, this task is performed using Loom's knowledge base retrieval technique. Specifically, a query is issued over the similar-to relation described in Section 2.4.2. Recall that this relation encodes a combination of both direct assertions and automatically computed inferences. For example, in the assembly example, there is no user-supplied assertion that assembly and disassembly are similar; rather, the similarity is computed from the fact that the makes conditions of these two tasks have similar logical structure and that each of the corresponding terms in these conditions either match or are directly related to each other (e.g., the assembled state in the makes condition of the assembly task is known to be the inverse of the disassembled state in the makes condition of the disassembly task).

After a known task has been retrieved, some sort of mechanism must be used to transfer the method for that existing task to address the new task. In REM, there is only one such mechanism: relation mapping. Relation mapping is a complex mechanism involving changes to several different tasks within the overall structure of ADDAM.

Algorithm **relation-mapping**(main-task, known-task)	
Inputs:	main-task: The task to be adapted
	known-task: A similar task with at least one method
Outputs:	main-task: The original task with a method added
Other Knowledge:	current-knowledge-state: The knowledge to be used by the task
Effects:	A new method has been created which accomplishes main-task within current-knowledge-state using a modified version of a method for known-task.

```
main-task:by-mmethod = COPY known-task:by-mmethod
map-relation = [relation which connects results of known-task to results of main-task]
mappable-relations = [all relations for which map-relation holds with some relation]
mappable-concepts = [all concepts for which map-relation holds with some concept]
relevant-relations = mappable-relations + [all relations over mappable-concepts]
relevant-manipulable-relations = [relevant-relations which are internal state relations]
candidate-tasks = [all tasks which affect relevant-manipulable-relations]
FOR candidate-task IN candidate-tasks DO
      IF [candidate-task directly asserts a relevant-manipulable-relations] THEN
         [impose the map-relation on the assertion for that candidate task]
      ELSE IF [candidate-task has mappable output] THEN
         [insert an optional mapping task after candidate-task]
RETURN main-task
```

Table 29: Algorithm for relation mapping

Furthermore, the process is not an entirely complete or reliable one; it involves suggesting a variety of modifications, but some of these changes are only tentative and may conflict with each other.

Table 29 presents the algorithm for relation mapping. This algorithm takes as input main-task and an existing known-task for which at least one method is already available (i.e., a task which the existing agent already knows how to perform). The relation mapping algorithm begins by copying the methods (including the methods' subtasks and those subtasks' methods) for known-task and then asserts that these copied methods are new methods for main-task; the remaining steps of the algorithm then modify these copied methods so that they are suited to main-task.

In the next step in the adaptation process, the system finds a relation which provides a mapping between the effects of main-task and the effects of known-task. In the assembly example, the relation which is used is inverse-of. This relation is selected by REM because (i) the task of disassembly has the intended effect that the object is disassembled, (ii) the task of assembly has the intended effect that the object is assembled, and (iii) the relation inverse-of holds between the assembled and disassembled world states. These three facts are explicitly encoded in the TMKL model of ADDAM. Note that there are other relations besides inversion which can be used for this mechanism; for example, some experiments with the monkey and tower agent described in Section 4.3 involve having the monkey retrieve fruit or plantains, using relation mapping on the specialization-of and generalization-of relations. The relation mapping algorithm only requires that a relation be a binary relation defined in either in the specific domain of the agent or in the general TMKL ontology; from these set of candidates one is chosen which maps values from the makes condition of the main task to the makes condition of the known task.

Once the mapping relation has been found, the next steps involve identifying aspects of the agent's knowledge which are relevant to modifying the agent with respect to that relation. The system constructs lists of relations and concepts for which the mapping relation holds. For example, ADDAM, being a hierarchical planner, has relations node-precedes and node-follows which indicate ordering relations among nodes in a hierarchical plan; these relations are the inverse of each other so both are considered mappable relations. A list of relevant relations is computed which contains not only the mappable relations but also the relations over mappable concepts. For example, the assembled and disassembled world states are inverse of each other (making that concept a mappable concept) and thus some relations for the world state concept are also included as relevant relations. This list of relevant relations is then filtered to include only those which can be *directly* modified by the agent, i.e. those which involve the internal

state of the agent. For example, node-precedes and node-follows involve connections between plan nodes which are knowledge items internal to the system. In contrast, the assembled and disassembled states are external. The system cannot make a device assembled simply by asserting that it is; it needs to perform actions which cause this change to take place, i.e., inverting the process of creating a disassembled state needs to be done implicitly by inverting internal information which leads to this state (such as plan node ordering information). Thus node-precedes and node-follows are included in the list of relevant manipulable relations while relations over world states are not.

Given this list of relevant manipulable relations, it is possible to determine the tasks which involve these relations and to modify these tasks accordingly. For example, one task in the ADDAM disassembly planning process directly asserts that a plan node precedes another plan node; this task is inverted in the assembly planning process to directly assert that the node follows the other node instead. Another example in ADDAM is the portions of the system which involve the types of actions in the plan (e.g., screwing is the inverse of unscrewing); in these situations, new tasks need to be inserted in the model to invert the output of those steps which produce this information. As noted in the previous section, these inserted tasks can conflict with each other so they need to be made optional; when the main-task is later executed, the inclusion or exclusion of these optional tasks is resolved through trial and error (via Q-learning).

Figure 30 presents the results of the relation mapping process over ADDAM in the assembly example, again focusing on those elements which are relevant to this discussion. After the disassembly process is copied, there are three major changes made to the copied version of the model:

1. The Make Plan Hierarchy process is modified to adjust the type of actions produced. Because the primitive tasks which manipulate plan nodes are implemented by procedures, REM is not able to directly modify them. Instead, it inserts a new mapping task in between the primitive tasks. The mapping task alters an action after it is created but before it is included in the plan. This newly constructed task asserts that the action type of the new node is the one mapped by the inverse-of relation to the old action type; for example, an unscrew action in an old disassembly plan would be mapped to a screw action in a new assembly plan.

2. The portion of the Map Dependencies process which asserts a dependency is modified. Because the primitive task for asserting dependencies is implemented as a simple logical assertion, it is possible to impose the inverse-of relation on top of that assertion. If one action was to occur before another action in the old disassembly plan then the related action in the new assembly plan occurs after the other action (because inverse-of holds between the relations indicating before and after). For example, if an old disassembly plan requires that boards be unscrewed before they can be removed, the new assembly plan will require that they be placed before they can be screwed together.

3. The execution process is modified to adjust the type of actions executed. This adjustment is done by an inserted task which maps an action type to its inverse. For example, if a screw action is selected, an unscrew action is performed.

Obviously the first and third modifications conflict with each other; if the system inverts the actions when they are produced *and* when they are used, then the result will involve executing the original actions. In principle, if the TMKL model of ADDAM were precise and detailed enough, it might be possible for REM to deduce from the model that it was inverting the same actions twice. However, the model does not contain the level of detail required to deduce that the actions being produced in the early portion of the process are the same ones being executed in the later portion. Even if the information were there, it would be in the form of logical expressions about the requirements and results of all of the intervening tasks (which are moderately numerous, since these inversions take place in greatly separated portions of the system); reasoning about whether a particular knowledge item were being inverted twice for this problem would be a form of theorem proving over a large number of complex expressions, which can frequently be intractable.

Fortunately, it is not necessary for REM's model-based adaptation technique to deductively prove that any particular combination of suggestions is consistent. Instead, REM can simply execute the modified system with the particular decisions about which modifications to use left unspecified. In the example, REM makes the two inserted mapping tasks optional, i.e., the state-transition machine for the modified methods has one transition which goes into the inserted task and one which goes around it. During execution, the decision making (Q-learning) process selects among these two transitions. Through experience, the decision making process develops a policy of including either of the inserted mapping tasks but not both.

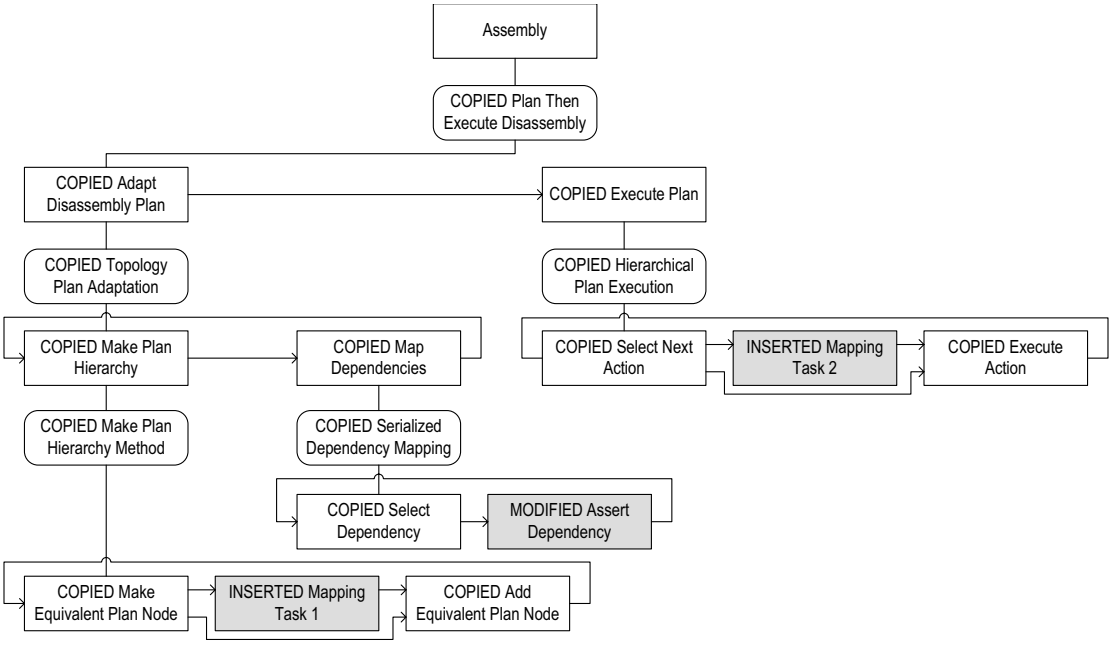


Figure 30: Tasks and methods produced for the assembly process by adapting ADDAM. Tasks which have been added or modified are highlighted in grey.

Note that REM is using exactly the same decision making component here that it uses to perform this task using the situated learning strategy; however, here this component is being used *only* to decide among the options which model-based adaptation left unspecified. In contrast, the situated learning algorithm uses the Q-learning to select from *all* possible actions at *every* step in the process. The Q-learning that needs to be done to complete the model-based adaptation process occurs over a much smaller state-space than the Q-learning for the complete problem (particularly if the problem is, itself, complex); this fact is strongly reflected in the results presented in Section 7.1.

6.4 Failure-Driven Model Transfer

Table 30 provides a broad overview of the failure-driven model transfer process within REM. There are three major components of this process: feedback analysis, failure selection, and repair. The input includes the task which was unsuccessfully addressed, the trace of the unsuccessful problem solving episode, and the feedback provided by the user (if any). Recall from Chapter 5 that if any failures were detected during execution, the trace includes annotations indicating what types of failures were encountered and where the failures occured. For failure-driven model transfer to be invoked, there must be a trace and there must either be at least one failure annotation on the trace or some feedback; if there is neither, then there is no evidence that there was a failure. The first step of the process involves analyzing the feedback in combination with the trace to detect any failures which were not evident at run time, i.e., failures which are only apparent due to the feedback. After this analysis, the process enters a loop. Inside the loop a failure is selected and then there is an attempt to repair that failure. The process continues until either some repair attempt is successful or there are no more failures to attempt.

Table 31 provides a more detailed view of feedback analysis (the first step of failure-driven model transfer). The process breaks the feedback into individual assertions and analyzes each one separately. For each assertion, the process first searches for elements of a trace which are potentially in conflict with that assertion and then annotates those trace elements with an indication of the conflict. The failure annotation may be of two different types: directly-contradicts-feedback and may-contradict-feedback. For example, if a piece of feedback indicates that a particular value should have been produced by the main task which was invoked, and some subtask of that main task which could produce that value doesn't, then a may-contradict-feedback annotation is placed because that subtask may have been the one that should have produced that value (but then again, some other subtask may have been the one which should have produced that value). If, on the other hand, the feedback specifically indicates a particular subtask which

Algorithm **failure-driven-model-transfer**(trace, feedback)	
Inputs:	trace: A trace of the execution of main-task
	feedback: Additional information, if any, provided by the user regarding the execution
Outputs:	main-task: The original task with some modification
Other Knowledge:	*none*
Effects:	Some portion of the main-task and/or its methods, subtasks, etc. has been modified to address a failure of the trace

```
analyze-feedback(main-task, trace, feedback)
REPEAT
    failure-annotation = select-failure(trace)
    [apply relevant repair mechanism if any for failure-annotation]
UNTIL [repair is successful] OR [there are no more failures]
RETURN main-task
```

Table 30: Main algorithm for failure-driven model transfer

should have produced a specific value for the given input, then a directly-contradicts-feedback annotation is produced because the feedback unambiguously indicates that this particular piece of the process was at fault. In addition to the type of failure, the annotation also contains the desired input and output knowledge states (as indicated by the feedback) and the actual input and output knowledge states (as indicated by the trace). Once the annotation is created, it is attached to the element of the trace for which it was produced.

The failure selection mechanism in REM is a fairly simple: each failure which has not yet been attempted is assigned a priority value and whichever failure has the highest priority is selected; if there is more than one failure with the highest priority, one is selected arbitrarily. The assignment of priorities to failures in REM is relatively *ad hoc* and is based primarily on the type of the failure. For example, directly-contradicts-feedback failures are prioritized over may-contradict-feedback failures because the former are more certain than the latter (so a repair to the former is more likely to allow the system to operate correctly than a repair to the latter). In general, prioritization of failures to repair is a challenging and important issue: this prioritization should be based on an estimation of the cost of repair and the likelihood that a repair attempt will produce the desired result. Because there are relatively few repair mechanisms in REM, a few simple heuristics address this problem adequately. However, it is to be expected that as new repair mechanisms are developed in future research, that the failure selection process will need to become increasingly sophisticated. Currently REM has only two repair mechanisms: generative planning and fixed value production. These mechanisms are described in the next two subsections.

6.4.1 Generative Planning Repair

The generative planning repair mechanism is the same process that is described in Section 6.2; in addition to the use described in that section (developing a method for a main task which has no method), this technique can also act as a repair strategy when the main task *or* a subtask of that main task has no applicable method in a particular situation, as indicated by the trace.

Figure 31 presents an ablated version of the monkey and tower agent which is described in Section 4.3. The ablated version has one of the methods for the Move Tower task removed so that only Trivial Strategy remains. Recall that Trivial Strategy has a provided condition that requires that there be only one disk. Thus the ablated version presented here will only execute successfully if there is one disk. If there are more than one disk, the execution of the agent fails when it attempts to select a method for Move Tower. At this point, a no-applicable-mmethod failure annotation is added to the trace of the Move Tower task and execution of the agent halts (as described in Section 5.3). Since the makes condition of the Move Tower task (that all of the disks be where the bananas are) has not been met, a makes-fails failure annotation is then added to that trace as well. In addition, the makes condition of the main Get Bananas task (that the monkey has the bananas) has also not been met, so a makes-fails annotation is

Algorithm **analyze-feedback**(trace, feedback)	
Inputs:	trace: A trace of the execution of main-task
	feedback: Additional information, if any, provided by the user regarding the execution
Outputs:	trace: The original trace, possibly with some additional annotations
Other Knowledge:	*none*
Effects:	Conflicts between what the trace indicates happened and what the feedback indicates should have happened are identified as failure annotations on the trace.

```
FOR feedback-term IN feedback DO
    FOR trace-element IN [search trace for potential contradictions to feedback-term]
        IF [feedback-term directly contradicts trace-element] THEN
            failure-annotation = NEW directly-contradicts-feedback annotation
        ELSE
            failure-annotation = NEW may-contradict-feedback failure annotation
        failure-annotation:desired-input = [input state of feedback]
        failure-annotation:desired-output = [output state of feedback]
        failure-annotation:actual-input = [input state of trace]
        failure-annotation:actual-output = [output state of trace]
        [attach failure-annotation to trace-element]
RETURN trace
```

Table 31: Algorithm for feedback analysis

also added to the higher-level trace for the main task.

Because there is at least one failure annotation, REM decides to invoke the failure-driven model transfer algorithm (from Table 30). The first step in that process is analyze-feedback (which is detailed in Table 31). In this example, there is no feedback, so the main loop in analyze-feedback, which steps through the terms of the feedback, is executed zero times, and thus no additional failures annotations are produced. The next step in the failure-driven model transfer algorithm is select-failure; in the example, the makes-fails annotation is selected. The only repair strategy that REM has for this type of failure annotation is generative planning repair so that process is performed, i.e., the generative planning algorithm described in Section 6.2 is invoked with the Move Tower task as its input.

The plan which is produced by Graphplan when the tower has two disks involves three disk movements: first it moves the little disk to the intermediate location, then it moves the big disk to the final location, then it moves the little disk to the final location. Figure 32 presents the tasks and methods of the adapted monkey and tower agent for the two disk example. The newly constructed method has three steps, all of which are the Move Disk subtask. Note that the transitions in the new method contain all of the information about the bindings of the parameters (in this case, which disk to move and where to move it). A description of how transitions are used to link parameter bindings appears in Section 2.3; information about how these bindings are produced for the planning adaptation strategy appears in Section 6.2.

6.4.2 Fixed Value Production

Unlike the generative planning repair mechanism, fixed value production is a model-based technique which is only applicable to failure-driven repair (because it relies specifically on the relationship between the trace and the feedback). SIRRINE also has two repair mechanisms, but both are variations of the fixed value production strategy. In this discussion we focus on the algorithms encoded in REM; for more on the two variations of fixed value production encoded in SIRRINE, see [Murdock and Goel, 1999a] and [Murdock and Goel, 1999c] respectively.

The overall purpose of the fixed value production repair strategy is to ensure that whenever the desired input values suggested by the feedback are provided that the desired output values suggested by the feedback are produced. The strategy also ensures that the behavior of the task being repaired is unaffected in all other cases. The algorithm for the fixed value production repair strategy is presented in Table 32. It takes as input the failure-annotation produced

Algorithm **fixed-value-production-repair**(failure-annotation)	
Inputs:	failure-annotation: a record of a task failure
Outputs:	*none*
Other Knowledge:	task-trace: The trace to which failure-annotation is attached
	original-task: The task for which task-trace records the execution
Effects:	An alternative method has been created for original-task which produces results indicated by failure-annotation.

original-task = [the task executed in the trace element for failure-annotation]
base-task = [copy of original-task]
alternative-task = NEW task
alternative-task:input = original-task:input
alternative-task:output = original-task:output
FOR binding IN failure-annotation:desired-output
    [add binding to the alternative-task:binds]
FOR binding IN failure-annotation:desired-input
    [add a check that binding holds to alternative-task:given]
base-method = NEW method
[create state machine for base-method that simply invokes base-task once]
alternative-method = NEW method
[create state machine for alternative-method that simply invokes alternative-task once]
alternative-method:provided = alternative-task:given
[delete implementation of original-task]
original-task:by-mmethod = [list of base-method and alternative-method]

Table 32: Algorithm for fixed value production repair

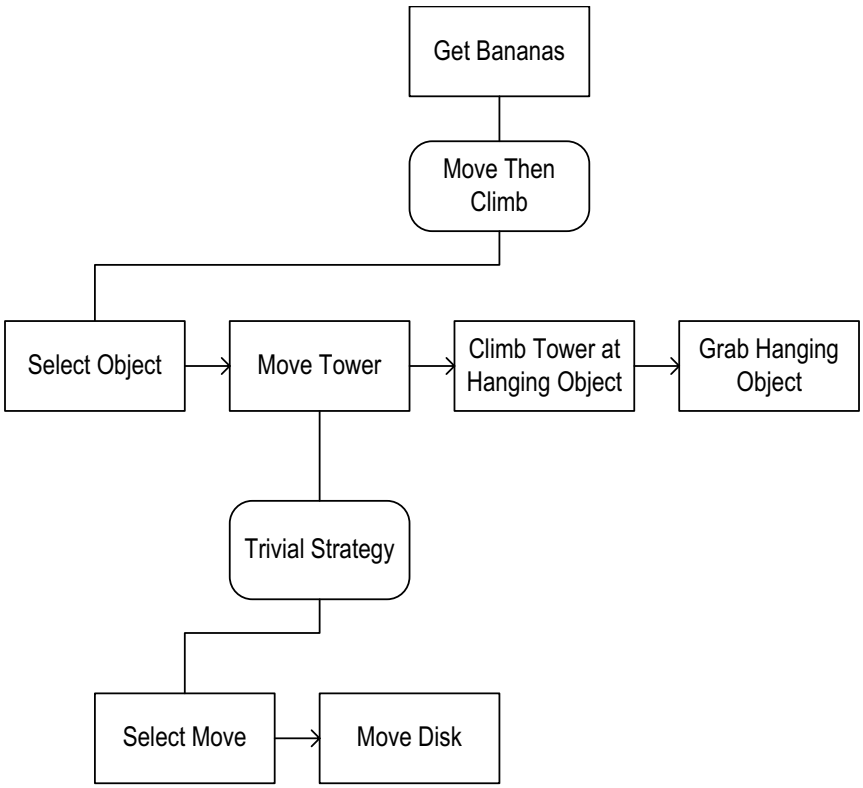


Figure 31: The monkey and tower agent from Section 4.3 with Nilsson's Strategy removed (i.e., the only method for Move Tower is Trivial Strategy).

by the failure selection mechanism. Note that this annotation is attached to some trace element and this trace element is attached to some portion of the TMKL model, so while the annotation is the only direct input, the strategy does make indirect use of both trace and model information. The process begins by accessing the original task which the annotation (indirectly) refers to. Next it makes a copy of that original task (including its implementation), which it refers to as base-task. After that, it generates a new task called alternative-task. The input and output parameters for that new task match those of the original task. The implementation of the new task takes the form of a binds slot which simply binds the parameters to the values indicated in the desired-output. The given requirement for the new task checks to see if the current state matches the bindings in the desired-input. Next two new methods are created: one which only invokes base-task and one which only invokes alternative-task: the latter is given a provided condition which matches the given condition of alternative-task. Lastly, the existing implementation for the original task is deleted and replaced with the two new methods. Because the alternative method has a provided clause which requires that the current state match the inputs from the feedback, when the original task is later executed it invokes that method in exactly that situation; in all other situations, the base task (which does exactly what the original task did before the repair) is invoked.

As an example, consider the following use of the web browsing agent from Section 4.2. The browser is asked to retrieve the document at the following URL:

http://thesis.murdocks.org/thesis.pdf

This document is an Adobe Portable Document Format (PDF) file of MIME type application / pdf. Mosaic 2.4, on which the web browser is based, has a simple, fixed set of MIME types for which it has an external viewer. The application / pdf type is not one of these. Consequently, the web browser is not able to display this file. In particular, the Select Display Command task does not produce any output. Consequently, one of the input parameters for the Compile Display Command is not available. Thus the execution process fails at that point, and a missing-inputs failure annotation is placed on the trace for Compile Display Command. Furthermore, makes-fails annotations are created for Compile Display Command each of the higher level tasks that it is included in (Display Interpreted File, Display File,

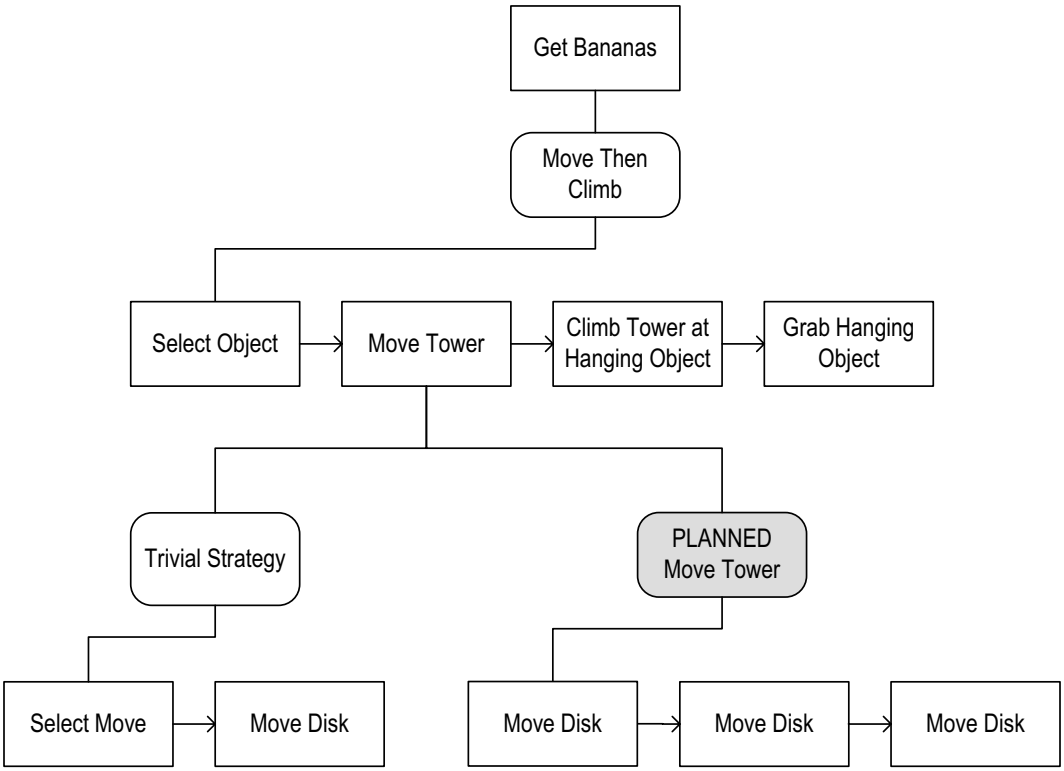


Figure 32: The ablated monkey and tower agent after being executed with two disks and then adapted using generative planning repair. The newly constructed method is highlighted in grey. Note that the model produced for more than two disks is similar but includes more Move Disk actions.

and Process URL). This is a total of five failure annotations.

After execution, feedback is received from the user. In this example, the feedback indicates that the should-transform relation holds over the main Process URL task, and an input state mapping the MIME tag to application / pdf, and an output state mapping the abstract command to "acroread ˜a". In other words, the user is explicitly stating that given an application / pdf tag, the agent should produce the "acroread ˜a" abstract command. Note, however, that the user is *not* stating what part of the agent should be responsible for producing this value. The feedback only mentions that the transformation needs to take place within the main Process URL task; it does not identify a specific lower level task which should be responsible for that change.

Because there are failure annotations and a feedback item, REM chooses to apply the failure-driven model transfer mechanism. Recall that the first step of that mechanism is analyze-feedback. The feedback analysis produces one additional failure annotation: a may-contradict-feedback annotation on the trace for the Select Display Command task. This is the only contradiction because this is the only task in the hierarchy which takes a tag as input and produces an abstract command as output. After feedback analysis, a failure annotations is selected. There are six such annotations to chose from in the example: the five which were generated during execution and the one which was generated during feedback analysis. The may-contradict-feedback is the one selected in this example. The repair strategy that REM has for this type of failure is fixed value production.

Figure 33 presents the tasks and the methods of the web browser after the application of fixed value production. The Select Display Command task, which was primitive in the original web browser, is divided into two methods. The base method contains a primitive task which does exactly what the old Select Display Command task did. The alternative method contains a primitive task which always outputs the same abstract command: "acroread ˜a". The provided condition for the abstract command requires that the input tag be application / pdf. Thus the modified TMKL agent is now capable of producing the appropriate command for PDF files (using the alternate method) but is also still capable of producing all of the commands that it could before (using the base method).

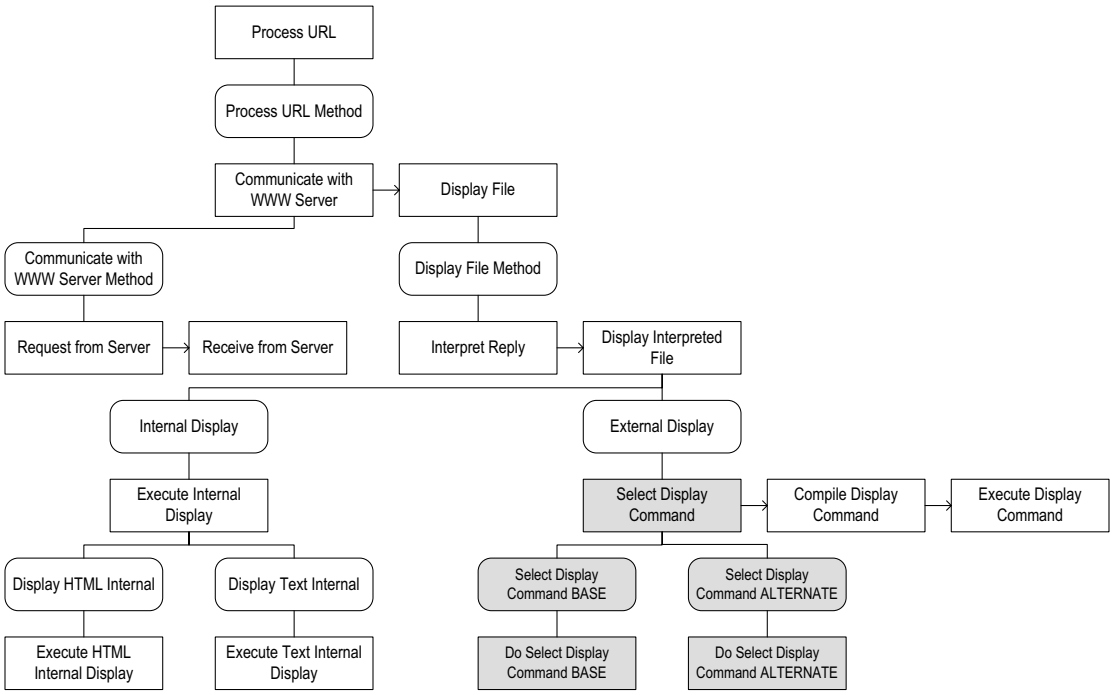


Figure 33: The web browsing agent after being executed for a PDF file and then adapted using fixed value production repair. The modified task and added tasks and methods are highlighted in grey.

The adaptation strategies presented in this chapter constitute the most significant aspect of the theory for this dissertation; the primary reason why models and traces of agents exist in REM and SIRRINE is to provide support for adaptation. Chapter 7 provides an evaluation of the reasoning shells using the example agents in Chapter 4. The focus of that evaluation is on the analysis of the effectiveness and efficiency of the adaptation strategies presented in this chapter.

Chapter 7

Evaluation

This chapter describes the techniques which were used to evaluate this work and the results provided by those techniques. The first section describes some computational experiments, i.e., specific occasions of the execution of the agents and algorithms described in the preceding chapters. The second section provides a more abstract analysis of the computational complexity of these techniques. The combination of these two forms of evaluation forms the basis for the discussion in the chapters that follow.

7.1 Experiments

There have been a wide variety of experiments involving REM and SIRRINE which have been conducted during the course of this research. This section describes the most significant experiments and results. Because REM is the latest and most advanced of the two reasoning shells, the bulk of the this section focuses on experiments with REM. The last subsection, however, does provide some results from SIRRINE.

7.1.1 ADDAM

The combination of REM and ADDAM has been on tested on a variety of devices. Some of these devices include a disposable camera, a computer, etc.[1] The nested roof example in Section 4.1.1 is one which has undergone particularly extensive experimentation in the course of this research. A key feature of the roof design from an experimental perspective is the variability in its number of components; Figure 18 in Section 4.1.1 shows the roof with five boards but it is easy to see how the design could be extended or retracted by adding or subtracting boards. This variability is useful for experimentation because it is possible to see how different reasoning techniques compare on the same problem at different scales.

[1]Note that those devices could also be disassembled by ADDAM alone. Thus, the ability for ADDAM to disassemble these devices within REM merely establishes that no functionality was *lost* in encoding ADDAM in TMKL and executing it within REM. This enhances the significance of the novel results which are described in the remainder of this section, in that these new capabilities are integrated into an agent which has demonstrated its effectiveness on a broad range of inputs.

The descriptions of the generative planning, situated learning, and proactive model transfer adaptation strategies presented in Sections 6.1, 6.2, and 6.3 all use the task of assembly as a motivating example. The first two of these sections involve the use of only the *primitive* knowledge items and tasks from the ADDAM domain. For example, the information available to REM in those two processes involves components (such as boards and screws) and actions (such as screwing and unscrewing), but does not include anything about ADDAM's reasoning techniques or abstract ADDAM knowledge such as hierarchical designs. In the process described in Section 6.3, REM did have access to the complete ADDAM TMKL model. In that process, REM adapts the design of the existing disassembly agent to address the assembly task. Thus the results of running these three processes provides an experimental contrast between techniques which not not make use of the existing model and those which do.

The performance of REM on roof assembly problem is presented numerically in Table 33 and graphically in Figure 34. The first series involves the performance of REM when uses ADDAM via proactive model transfer. The other two data series involve the performance of REM without access to the full ADDAM model; these attempts use generative planning (based on Graphplan) and situated learning (based on Q-learning), respectively.[2] The key observation about these results is that both Graphplan and Q-learning undergo an enormous explosion in the cost of execution (several orders of magnitude) with respect to the number of boards; in contrast, REM's proactive transfer (with assistance from Q-learning) shows relatively steady performance.

The reason for the steady performance using proactive transfer is that much of the work done in this approach involves adapting the agent itself. The cost of the model adaptation process is completely unaffected by the complexity of the particular object (in this case, the roof) being assembled because it does not access that information in any way; i.e., it adapts the existing specialized disassembly planner to be a specialized assembly planner. The next part of the process *uses* that specialized assembly planner to perform the assembly of the given roof design; the cost of this part of the process is (obviously) affected by the complexity of the roof design, but to a much smaller extent than generative planning or reinforcement learning techniques are.

Boards	REM with ADDAM model	REM with Generative Planning	REM with Situated Learning
1	36.3	6.2	6
2	82.7	6.5	92.7
3	156	6.9	297000.2
4	255.7	8.2	
5	381.1	665	
6	541.5	*106063.8*	
7	721		

Table 33: Elapsed time (in seconds) taken using different techniques within REM on the roof assembly example (which involves many goal conflicts) for a varying number of boards. The italicization of the last item in the Graphplan series indicates abnormal termination (see text). This data is also presented graphically in Figure 34.

In a related experiment, the same reasoning techniques are used in this experiment but the design of the roof to be assembled is slightly different: specifically, the placement of new boards does *not* obstruct the ability to screw together previous boards. These roof designs contain the same number of components and connections that the roof designs in the previous problem do. However, planning assembly for these roofs using generative planning is much easier than for the ones in the previous experiment because the goals (having all the boards put in place and screwed together) do not conflict with each other. Table 34 and Figure 35 shows the relative performance of the different approaches in this experiment. In this experiment, REM using only Graphplan is able to outperform the model transfer approach; i.e., because the problem itself is fundamentally easy, the additional cost of using and transforming a model of ADDAM outweighs any benefits that the specialized assembly planner provides. This illustrates an important point about model-based adaptation: i.e., that it is ideally suited to problems of moderate to great complexity. For very simple problems, the overhead of using pre-compiled models can outweigh the benefits.

The results of these and other experiments have shown that the combination of model-based adaptation and reinforcement learning on ADDAM provides tractable performance even for devices which cannot be handled effectively

[2]There is one flaw in the Graphplan series which is noted on both the table and the graph: for the six board roof, Graphplan ran as long as indicated but then crashed, apparently due to memory management problems either with Graphplan or with REM's use of it.

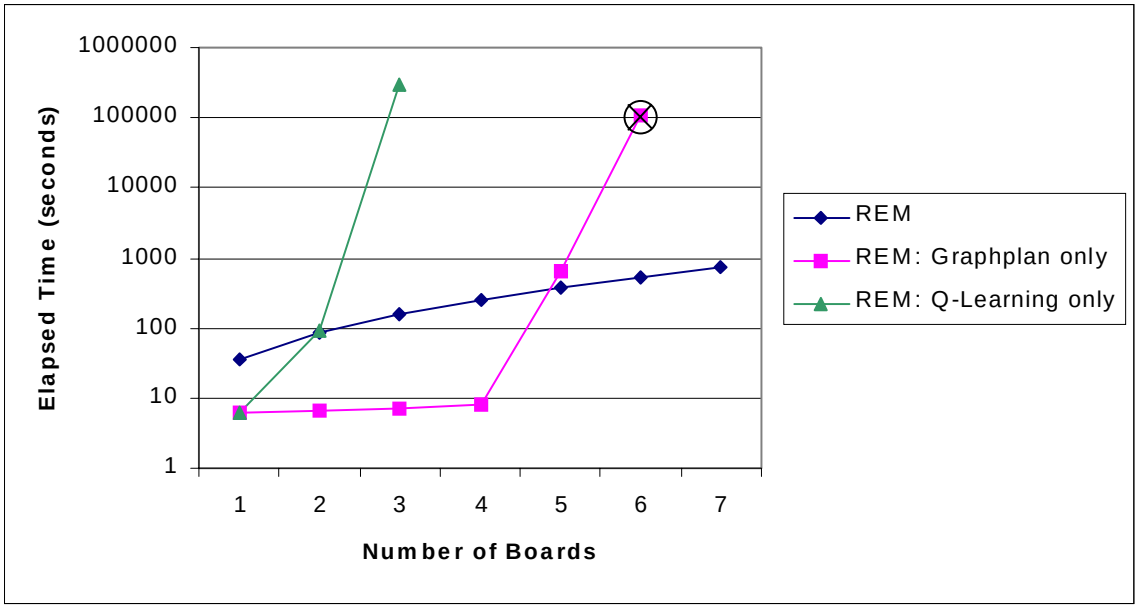


Figure 34: Logarithmic scale graph of the relative performances of different techniques within REM on the roof assembly example (which involves many goal conflicts) for a varying number of boards. The "X" through the last point on the Graphplan line indicates abnormal termination (see text). This data is also presented numerically in Table 33.

Boards	REM with ADDAM model	REM with Generative Planning	REM with Situated Learning
1	36.6	6	5.9
2	83.8	6.4	88.9
3	154.8	6.9	2190.2
4	257	7.2	90536.4
5	380.6	8	
6	543.1	8.8	
7	720.6	9.4	

Table 34: Elapsed time (in seconds) taken using different techniques within REM on a modified roof assembly example in which there are no conflicting goals. This data is also presented graphically in Figure 35.

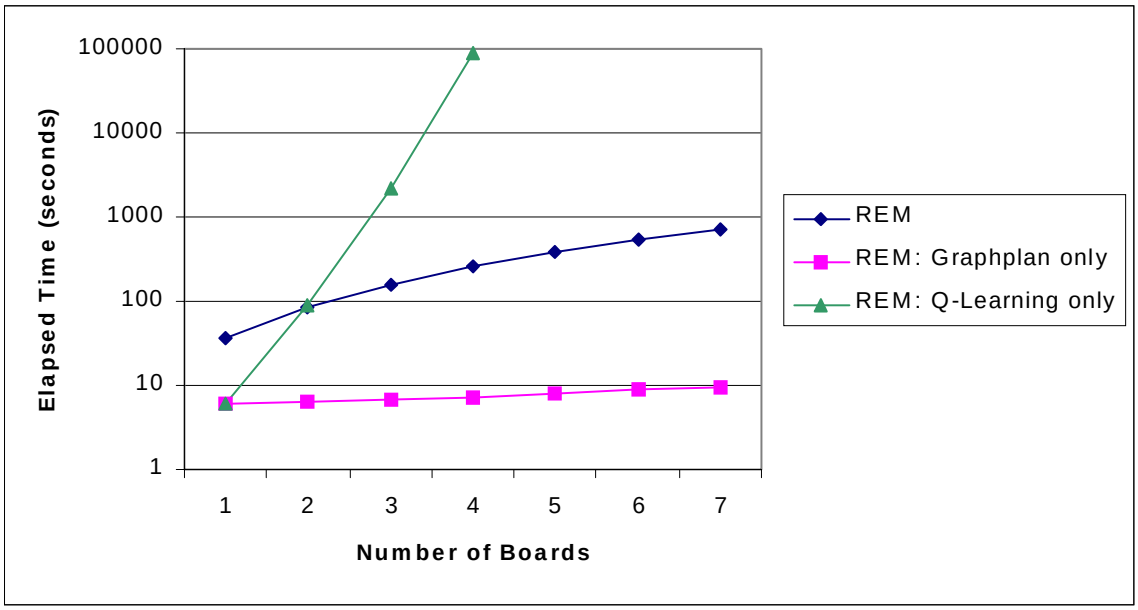


Figure 35: Logarithmic scale graph of the relative performances of different techniques within REM on a modified roof assembly example in which there are no conflicting goals. This data is also presented numerically in Table 34.

by the alternative approaches. The pure reinforcement learning approach is overwhelmingly expensive for all but the most trivial devices. The Graphplan approach is extremely rapid for relatively simple devices and even scales fairly well to *some* devices which contain substantially more components. However, there are other, very similar devices which involve more complex relationships among the components for which Graphplan is completely overwhelmed. The adapted ADDAM system is able to handle devices of this sort much more quickly than Graphplan is.

7.1.2 The Web Browsing Agent

Section 6.4.2 presents an experiment with the web browsing agent in which the agent is given the URL for a PDF document but does not have an external viewer which is capable of displaying files of this type. This experiment is motivated by the fact that the addition of new external viewers to the web browser is a recognized real-world problem. Recall that the web browsing agent is intended to be a mock-up of the Mosaic for X-Windows 2.4 web browser. The Mosaic "Wish List" [NCSA, 1997, p. 12] includes "on-the-fly selection of external viewers" as a desired improvement to Mosaic. The PDF example addresses a limited version of this goal, i.e., adding a single specific new external viewer.

Obviously it is possible for a non-intelligent web browser to include a built-in mechanism for automatically configuring new external viewers; most recent commercial web browsers have this capability. Indeed any feature that one could develop through model-based adaptation could, instead, have been encoded in the original system. However, a sufficiently dynamic environment will always impose new demands.

Gerhard Fischer [2001, p. 11] defines "adaptable" systems as ones in which the "user changes (with substantial system support) the functionality of the system." Fischer contrasts this with the notion of "adaptive" systems which involve "dynamic adaptation by the system itself to the current task and user." One of the key results of the PDF experiment is that it establishes the effectiveness of this theory in enabling adaptable behavior. In the PDF example, the user knows precisely what results the agent should provide, and provides this information to the agent via feedback; the contribution made by the reflective agent in this case involves finding a particular portion of itself to modify and making the modification which produces the value given in the feedback. In contrast, the ADDAM assembly example discussed in the previous section demonstrates adaptive behavior; the user provides only a description of the overall task and the agent adapts itself to satisfy that task. In the latter example, the user does not provide any specific values or actions that the agent should produce or perform. The combination of the web browser / PDF and ADDAM / assembly experiments prove that the reasoning architecture developed in this research is applicable to both adaptable reasoning and adaptive reasoning.

7.1.3 The Monkey and Tower Agent

The work on the monkey and tower problem in REM was motivated primarily by a desire for a simple example to explore certain kinds of adaptation issues which do not happen to arise in the more substantial examples. For example, the use of generative planning as a repair strategy in failure-driven model transfer (as described in Section 6.4.1) has not been relevant to the problems which have been addressed in ADDAM and the web browser. The fact that REM is able to address this "toy" problem is not, in and of itself, a substantial validation of the theory behind REM. However, the existence of this agent in TMKL and its use of REM adaptation techniques to enhance itself do demonstrate some additional breadth of applicability for REM. Furthermore, the fact that numerical results with this agent (below) confirm trends that are found in the ADDAM example, shows that these trends are not merely artifacts of the ADDAM domain but rather that they have some degree of generality.

Table 35 presents the total time used for the problem of obtaining the banana in a variety of different situations with a tower which has a varying number of disks; this data is also portrayed graphically in Figure 36. The first data series shows the performance of the full agent presented in Section 4.3; because this agent has methods which work for any number of disks, no adaptation of any sort is needed. The second series shows the performance of the ablated agent which appears in Figure 31 of Section 6.4.1. Recall that this ablated agent is missing one of the methods for one of its subtasks: Move Tower. The agent is able to perform correctly when there is only one disk, but it fails when it encounters more than one disk. When it does fail, generative planning repair is used to build an additional method for Move Tower. The third and fourth series show performance when there is no use of an existing agent at all: the third series involves pure generative planning (i.e., generative planning for the entire main task) while the fourth series involves situated learning.

Disks	Full Agent	Ablated Agent	Generative Planning	Situated Learning
1	6.5	6.3	7.4	27.5
2	7.2	10.2	8.1	96.6
3	9.5	11.6	9.9	209.6
4	13.3	3653.4	15899.7	31915.9
5	21			
6	42.8			
7	108.4			

Table 35: Elapsed time (in seconds) taken by REM for the monkey and tower Get Bananas task using different mechanisms with different knowledge conditions. This data is also presented graphically in Figure 36.

There are several key insights which can be observed from Figure 36. One insight is the fact that the curves which involve planning and learning undergo a sudden, dramatic jump in cost even at a relatively small problem size. In contrast, the technique which uses only specialized reasoning techniques encoded in the model shows relatively steady performance. This contrast is very similar to the contrast in the ADDAM / roof experiments; Figure 36 bears a general resemblance to Figure 34 earlier in this chapter. One significant difference between these two graphs, however, is that even the best performance in the monkey and tower agent is obviously exhibiting at least exponential cost (as demonstrated by the upward concavity of the curve in the logarithmic plot). This is an inevitable consequence of the well-known fact that the Tower of Hanoi problem (which is part of the monkey and tower process) requires an exponentially increasing number of disk movements. Even if no time was needed to compute the actions, just performing them (in simulation) must take exponential time. This fact illustrates an important point about REM: the performance of REM running an existing agent is limited by the performance of that agent. If REM is given an exponentially expensive agent to run, it will be at least exponentially expensive. This issue is explored in more detail in the complexity analysis presented in Section 7.2. Here note that while the full monkey and tower agent does have an exponential cost, it is still dramatically faster for the larger numbers of disks than the alternative approaches which do not use the full agent.

The most interesting aspect of this data is the relative performances of the two series which involve generative planning (the second and third). The performance data for these two agents are similar to each other but not identical. For one disk, the ablated agent is slightly faster than pure generative planning. The ablated agent does not need to do any adaptation for one disk because the method that it does have can handle one disk. The pure

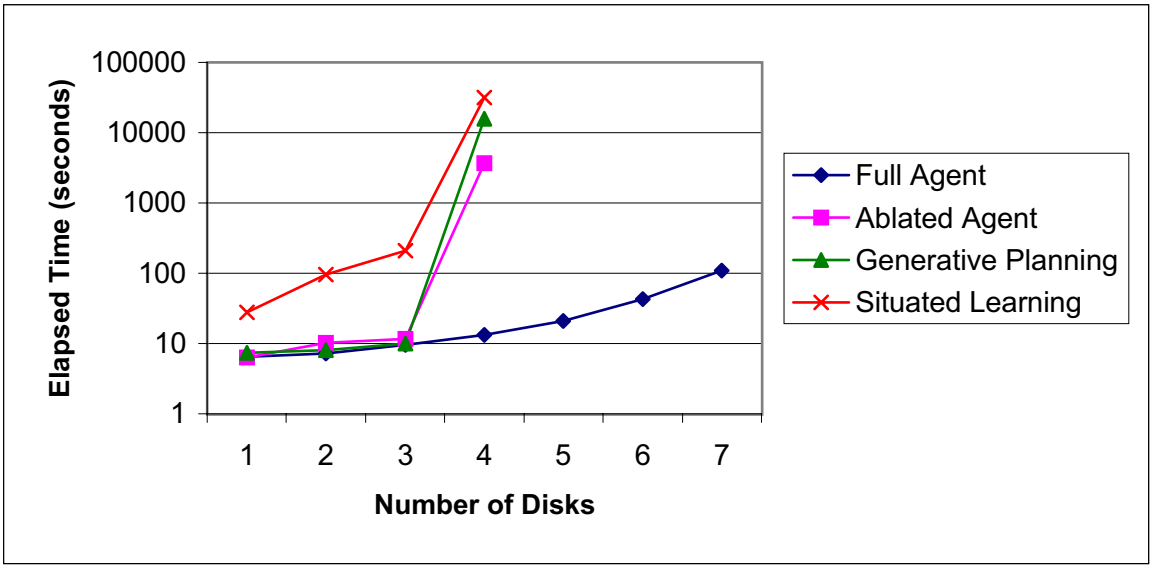


Figure 36: Logarithmic scale graph of the relative time taken by REM for the monkey and tower Get Bananas task using different mechanisms with different knowledge conditions. This data is also presented numerically in Table 35.

generative planning process is also very quick for the one disk case, but not quite as fast as the agent which already knows how to solve the problem.

For two and three disks, the performance of the ablated agent is slightly worse than the performance using pure generative planning. Whenever the number of disks is greater than one, the process for the ablated agent involves running the agent, encountering a failure, selecting a particular localization of the failure (as identified by trace annotations), planning at that localization, and then executing the corrected agent. Figure 37 (which also appears in Section 6.4.1) shows the ablated agent after adaptation in the two disk example. In contrast, the pure generative planning process involves just planning and executing. Figure 38 shows the pure generative planning agent after adaptation in the two disk example. Note that the process for the ablated agent involves a more complex model, plus an additional execution step, plus localization. These factors clearly impose some additional cost over the pure generative planning agent (although it does appear from the data that this cost is fairly small). Of course, the ablated agent does have one advantage: in the planning step, the ablated agent only needs to form a plan for moving the tower. In contrast, pure generative planning needs to plan the entire banana acquisition process (involving not only moving the tower but also climbing and grabbing bananas).

In the two and three disk examples, the advantage that the ablated agent has in addressing a simpler planning problem appears to be outweighed by the additional processing that it performs. However, in the four disk example the balance is shifted. Both processes show a dramatic jump in cost at the fourth disk. Note, however, that the cost for the ablated agent is lower by a noticeably wider margin than the advantage that the other approach had at two and three disks (even on the logarithmic scale graph). Specifically, the the ablated agent took approximately one hour while the pure generative planning agent took more than four and a half hours. Recall that that the only extra steps that need to be included in the plan for the pure generative planning agent are climbing and grabbing. In the smaller examples, the entire process involving moving disks, climbing, and grabbing is performed by the pure generative planning agent in under ten seconds. However, in the four disk example, planning these extra steps adds more than three and a half hours to the overall process, more than quadrupling the total cost. The monkey and tower problem is easily decomposed into moving the tower, climbing, and grabbing; however, the pure generative planning mechanism has no advance knowledge of that decomposition and must be constantly considering the issues involved in those extra steps throughout dealing with the tower issues. In contrast, the ablated agent already has the problem decomposed via its TMKL model. It is thus able to use the Graphplan only on the portion of the process that it does not already know how to solve. This experiment has shown that decomposition and localization via TMKL provide a substantial advantage for planning.

One issue which is not addressed in the data presented in Table 35 and Figure 36 is a comparison between pure model-based adaptation and other adaptation techniques. All of the variations in that data set involve the same main

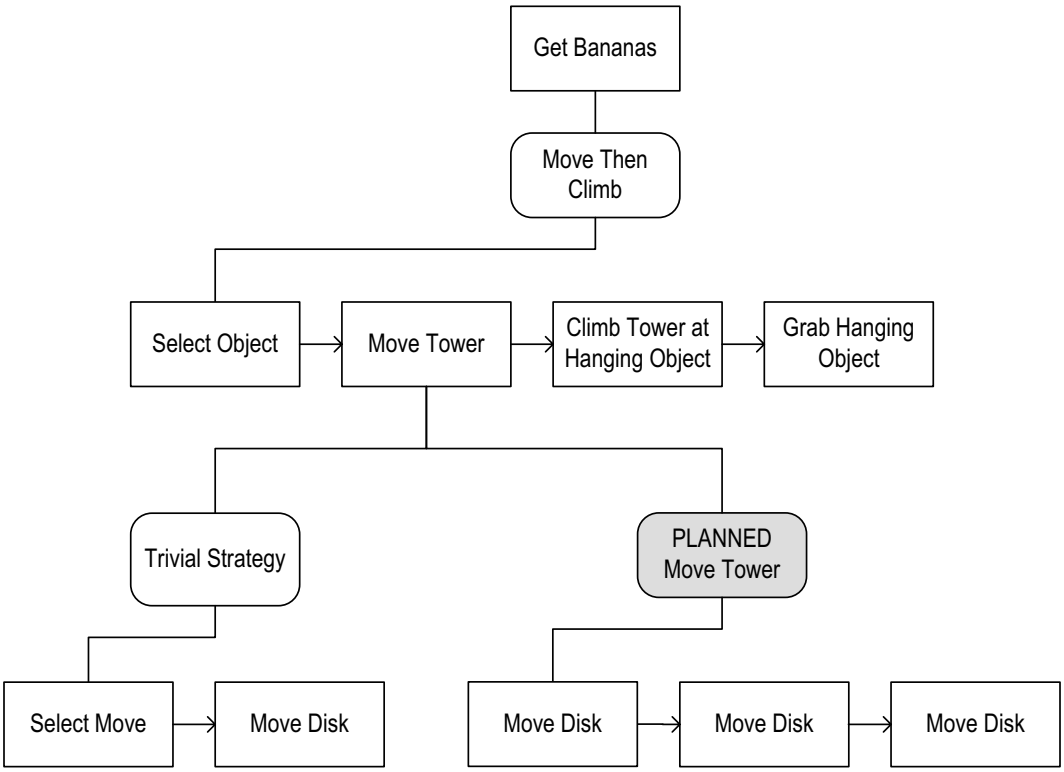


Figure 37: The ablated monkey and tower agent after being executed with two disks and then adapted using generative planning repair. The newly constructed method is highlighted in grey. Note that the model produced for more than two disks is similar but includes more Move Disk actions.

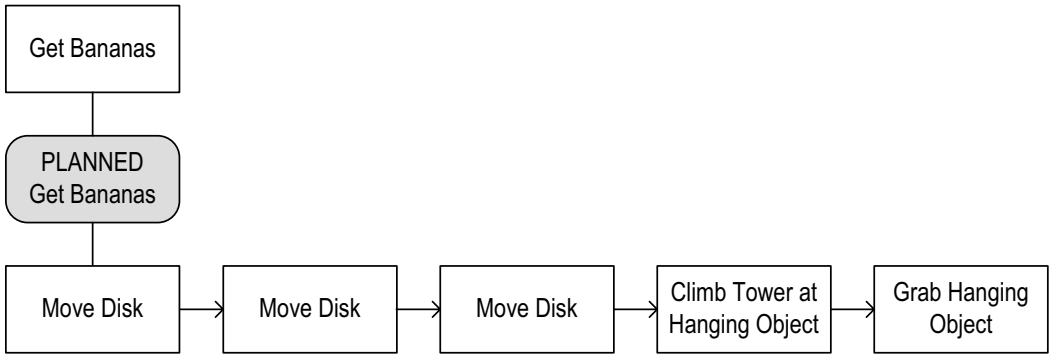


Figure 38: The monkey and tower problem after being executed with two disks and no pre-existing model; pure generative planning is used to produce the adapted agent. The newly constructed method is highlighted in grey. Note that the result for more than two disks is similar but includes more Move Disk actions.

task: getting bananas. The experiments with this task have not involved any knowledge conditions which demand pure-model based adaptation. It is possible, however, to observe the performance of pure model-based adaptation in transferring a model for that task to a different task. One such example which has been performed in REM involves getting fruit. The only difference between getting bananas and getting fruit is that the latter task has a more general requirement and does not have a pre-existing method within the monkey and tower agent. The background knowledge that the agent has does include the fact that the specialization-of relation maps bananas to fruit. Thus when REM attempts to address the task of getting fruit, it first notices that there are no available methods, then it retrieves the similar task of getting bananas, then it uses the relation mapping algorithm (from Section 6.3) to adapt the method for the banana task into a method for the fruit task.

Table 36 and Figure 39 present data which contrasts the results of using the full agent for the banana task with the results of using the same agent on the fruit task via proactive transfer (specifically, relation mapping). Note that the first data series is the same one that appears in the previous data set: the full agent addressing the banana task. The second series involves using the same agent on the fruit task. The most significant property of this data is the fact that while proactive transfer does add some additional costs, those extra costs are very small compared to the total costs for this problem. In the graph, the two curves appear to be coming together because of the logarithmic scale; the small, roughly constant difference in costs becomes increasingly less significant at the higher end of the scale. Note that this very small extra cost for pure model-based transfer is in dramatic contrast with the enormous extra costs imposed by generative planning and reinforcement learning in the data presented earlier in this section.

Disks	Full Agent	Proactive Transfer
1	6.5	8.8
2	7.2	9.8
3	9.5	11.9
4	13.3	15.4
5	21	24
6	42.8	45.3
7	108.4	113

Table 36: Elapsed time (in seconds) taken by REM using the full agent on the Get Bananas task versus using that same agent plus proactive model-transfer to address the Get Fruit task. This data is also presented graphically in Figure 39.

Several other experiments were run with the monkey and tower agent for which quantitative results were not recorded. In one of these experiments, the task was to get plantains, and the generalization-of relation was used to map bananas to plantains. The behavior observed in that experiment was similar to that in the fruit experiment. In yet another experiment, the ablated agent was first used to get bananas using generative planning repair and then was later asked to get fruit; relation mapping on the ablated but repaired model allowed the agent to reuse the plan for the Move Tower task in the context of getting fruit, even though this plan was originally generated for the banana task. This last experiment shows the cumulative nature of adaptation in REM: a model which is adapted for one purpose can later be further adapted for other purposes.

The two collections of data and the additional experimental results presented in this section demonstrate that the use of TMKL models in the monkey and tower problems does provide a significant benefits. This result provides additional validation for the similar result found in the ADDAM experiments described in Section 7.1.1. Furthermore, the monkey and tower experiments also contrast generative planning localized by TMKL models to generative planning without models; this contrast shows that (in this domain) generative planning without models is sightly faster for very simple problems but generative planning with models is much faster for larger problems. The complexity analysis of this mechanism in Section 7.2.2.4 suggests that this result is applicable across a variety of problems for which generative planning is very expensive.

7.1.4 Other REM Experiments

Another agent which exists within REM is a traditional monkey and bananas problem agent. Compared to the agent in the previous section, this agent has a slightly more detailed representation of the portions of the problem relating

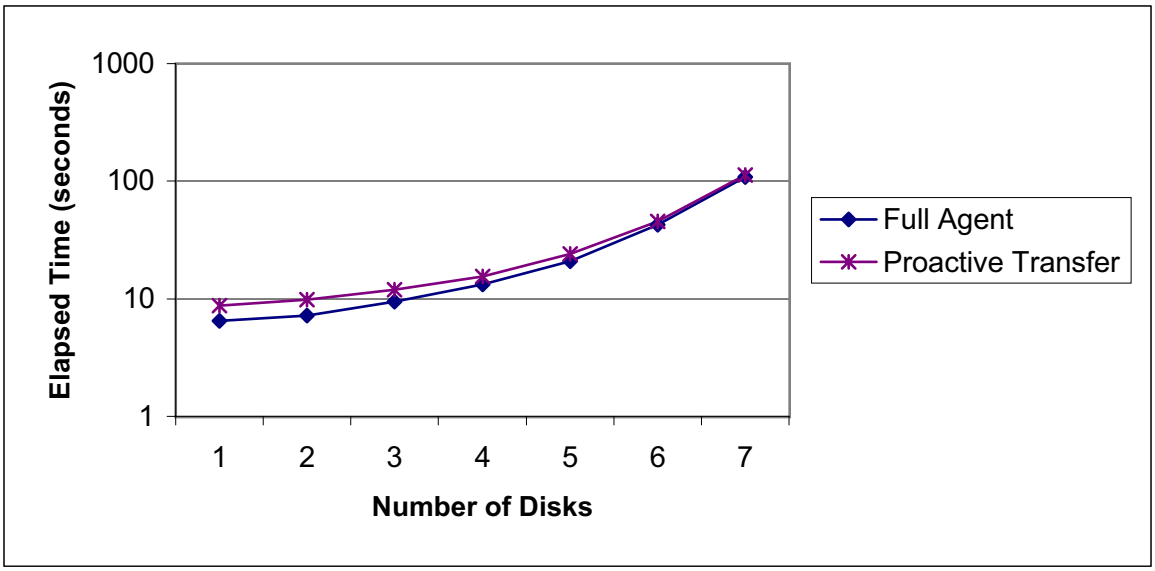


Figure 39: Logarithmic scale graph of the relative time taken using the full agent on the Get Bananas task versus using that same agent plus proactive model-transfer to address the Get Fruit task. This data is also presented numerically in Table 36.

to the monkey and the bananas, but does not have any of the Tower of Hanoi aspects of the problem. The primary point of that example is that it establishes the effectiveness of the REM generative planning adaptation strategy on a traditional, off-the-shelf planning problem.

The work on this and other agents[3] helps to establish the breadth of this research. This breadth adds to the significance of the experimental results established by the more elaborate experimental agents. Specifically, these results show that the usefulness of REM's execution and adaptation algorithms is not limited exclusively to those agents. Because the experiments with those agents involve substantial problems (such as assembly and browser configuration) which are known to have real-world importance, REM's ability to address these problems is inherently significant. The fact that the representations and processes which REM uses for these problems are also applicable to other kinds of problems proves that the effectiveness of the theory is not merely an artifact of the specific domains.

7.1.5 SIRRINE Experiments

Experiments with SIRRINE also provide significant validation of overall theory presented in this dissertation. One agent which was encoded in SIRRINE is the meeting scheduler described in Section 4.4. In one experiment with this meeting scheduler, there was an attempt to schedule a 90 minute meeting for which every slot that it considered conflicted with at least one schedule; consequently, the meeting scheduler failed to generate a time for the meeting. Feedback then indicated that the meeting would be held on Tuesdays from 4:30 to 6:00 PM, i.e., the assumption that meetings must be held during standard business hours was violated. At this point, the system was able to do some self-adaptation so that it would have generated this answer in the first place.

There are many possible changes that can be made which would have lead to this result. The meeting scheduler could be changed to always schedule meetings on Tuesdays 4:30 to 6:00 PM. Alternatively, it could be made to schedule meetings on Tuesdays 4:30 to 6:00 PM only when it receives exactly the same set of schedules as it did in the experiment and to simply use its existing mechanisms in all other cases. A reasonable compromise would be to allow meetings to generally be scheduled until 6:00 PM. However, the current model of the meeting scheduling

[3]The remaining REM agents are all incomplete or no longer functional. A "hello world" agent which was created early in the REM development process to explore how certain features of TMKL and Loom interact with each other; this agent was not maintained during the further development of REM and thus is incompatible with the final version of the shell. Some work was done on providing a navigational path planning agent based on Router [Goel et al., 1994], the employee time card system from ZD [Allemang, 1997], and another traditional planning domain. Obviously, because these agents do not currently work in REM, they do not directly provide additional validation of this work. However, they are mentioned here briefly to further illustrate some kinds of topics for which limited experience suggests that REM may be applicable.

domain and process does not provide sufficient information to identify a reasonable compromise. Consequently, a simpler change was made in which the Tuesdays 4:30 to 6:00 slot is suggested whenever a 90 minute time slot is requested and no other time is available. The mechanism which SIRRINE used to accomplish this is its version of the failure-driven model transfer mechanism described in Section 6.4 (specifically, it used the fixed value production repair strategy described in Section 6.4.2).

Another agent which was the subject of some experiments in SIRRINE was a web browsing agent which was a predecessor of the web browser in REM. The SIRRINE version has essentially the same behavior as the REM version but has a slightly less developed TMK model. One of the examples which was explored using the web browser in SIRRINE was the new external viewer problem which was also considered in REM. The mechanism by which SIRRINE accomplishes this is similar to the mechanism in REM. The main failure-driven model transfer algorithm presented in Section 6.4 is essentially the same in both SIRRINE and REM (there are, however, some differences in how these algorithms are implemented in the two shells). The repair strategy that SIRRINE employs for this example is a specialized variation of fixed value production. In this variation, no new tasks or methods are added to the agent; instead, the mapping from the MIME type to the abstract command is added to a look-up table which directly implements the task (this table is essentially a restricted form of the binds slot in a TMKL task). This specialized variant of fixed value production is more elegant than the more general version because it accomplishes the same effect with a much smaller modification to the agent. Of course, the drawback of the specialized variant is that it only works on tasks which are implemented by a look-up table. A task which is implemented by a method or a LISP procedure can not be modified in this way. SIRRINE has both the specialized variant and the more general version (i.e., the one described in Section 6.4.2); it uses the specialized one whenever possible and the more general one in all other situations. In contrast, REM only uses the more general version.[4]

In another SIRRINE web browser experiment, a URL is provided for a document which does not exist on the indicated server. Consequently, the attempt to retrieve the document from the server fails. In this experiment, user feedback is provided which specifies appropriate default behavior for this URL: the browser displays a special HTML file indicating that the desired document is not found. That special HTML file is treated as a specific value which the document retrieval task should produce. The document retrieval task is not implemented by a look-up table; such a table would need to contain all of the URL's and documents on the web! Therefore the specialized variant of fixed value production is not applicable. Consequently, SIRRINE uses the more general form of this repair strategy.

There were several additional experiments which were run on the web browser in SIRRINE. Some of these simply involved running the web browser on URL's for which the browser is able to successfully retrieve and display a document. These experiments proved that the SIRRINE web browser was able to perform correctly for ordinary requests which do not require any adaptation. The final two experiments each involved using the web browser on two different URL's in separate executions. In one of these experiments, the browser is run first on the URL for the PDF document and then on the URL for the non-existent document. In the other, the order is reversed. In both cases the end result is an adapted version of the browser which supports *both* PDF browsing and appropriate default behavior. These experiments explore the cumulative nature of agent adaptation: an agent which has been enhanced in one way from one problem may be further enhanced in another way from another problem.

7.2 Analysis

There are two key challenges in performing a complexity analysis of the algorithms presented in this dissertation. One of these challenges is the fact that the individual steps of the algorithms are, themselves, computational processes, some of which do not complete in a constant amount of time. For example, the first step of the execute-task algorithm involves the creation of a new method trace object. How much time this takes depends on the nature of the knowledge representation framework, how it builds new instances, how it stores and indexes them, etc. This challenge is addressed in this analysis by making some reasonable explicit assumptions about the costs of various kinds of steps. The other major challenge is the fact that the inputs to these processes can be intricate and involve many aspects, some of which can demand potentially unlimited computation. For example, a primitive task in a TMKL model can have an arbitrary LISP procedure as an implementation: it is well known that an arbitrary

[4]The specialized form of fixed value production was not included in REM, in part because the work in REM on adaptation focused primarily on new kinds of adaptation such as proactive model transfer, generative planning, etc. Because the binds slot in REM's TMKL serves a similar role to the look-up tables in SIRRINE's variant of TMK, it would certainly be possible to translate the specialized variant of fixed value production to work in REM using the binds slot. Note that while the specialized variant is more elegant, the effect that it has on the behavior of the agent is identical so the omission of the variant in REM does not affect the range of adaptation problems that it is able to address.

LISP procedure may not terminate, so it is impossible to place an upper bound on the cost of executing such a procedure. This challenge is addressed in this analysis by defining terms for those quantities which can not be further decomposed.

There are a variety of assumptions that are made about the costs of various kinds of steps of the algorithms. The cost of selecting one of a set of n alternatives is assumed to be $O(n)$; this is reasonable if the selection is done by a relatively straightforward computational mechanisms such as Q-learning (as is the case in REM). The total cost of retrieving a list of all values of a particular type is assumed to be at most linear in the number of values; e.g., if there are n tasks in a model, then retrieving a list of tasks is $O(n)$. It is also assumed that if a given value which instantiates n concepts, those concepts can be accessed in $O(n)$ time. Similarly, it is assumed that if a relation has n concepts that it relates to, those concepts can also be accessed in $O(n)$ time. Finally, it is assumed that the following sorts of computations are $O(1)$:

- Creating a new instance of a concept
- Accessing the value of a slot in an instance
- Setting the value of a slot in an instance
- Adding a new value to a slot in an instance
- Deleting a value from a slot in an instance
- Accessing the value of a parameter within a knowledge state
- Setting the value of a parameter within a knowledge state
- Adding a new term to a logical assertion
- Accessing the concept which a parameter instantiates
- Accessing a concept, instance, or relation given its name
- Returning a value from a call to an algorithm

One open question is whether these assumptions hold for REM. Most of the capabilities about which assumptions are made are basic knowledge representation operations like setting a slot in an instance. In REM, these operations are mostly handled by Loom. A complexity analysis of all of the operations of Loom is not available, so it is not possible to make a certain claim about whether REM adheres to all of the given assumptions. In particular, Loom does some truth maintenance in response to many kinds of operations and it is difficult to say how expensive those processes are. However, it does seem fairly plausible that Loom comes reasonably close to adhering to the given assumptions in most of the situations for which REM has been used.

The terms which are used in the analysis are presented in Table 37. The two most significant terms presented there are s and t: the total sizes of a model and a trace, respectively. The remaining terms provide more detail about quantities in the model and the input. The last four terms specifically describe certain kinds of costs which cannot be bounded using the quantities which appear above it.

7.2.1 Complexity of Execution

There are two pieces of the cost of execution which have no upper limit on their costs: C_e, the cost of executing primitives, and C_l the cost of checking logical assertions. TMKL provides a sufficiently powerful formalism for both primitive tasks and logical assertions that either of these values can have no limit. The execution algorithms presented in Chapter 5 involve performing these operations so their total cost must certainly be at least $C_e + C_l$. It is possible to consider how much additional costs beyond $C_e + C_l$ are imposed by this algorithm. The total costs not involving $C_e + C_l$ for individual steps and for the algorithms as a whole are indicated in boldface below (to make it easier to see what pieces are totaled up in the end).

The task execution algorithm appears in Table 38; this algorithm also appears in Section 5.1, where it is explained in more detail. The algorithm executes once for each task invocation. The first few steps involve creating a task trace and setting its values; recall that these operations are assumed to be $\boldsymbol{O(1)}$. Next there is a check to see if the parameters are bound; this check involves at most m_p parameters and each one requires an access to a parameter

Term	Description
s	Total number of elements of any sort in a model
t	Total number of elements of any sort in a trace
n_t	Total number of tasks in a model
n_c	Total number of concepts in a model
n_r	Total number of relations in a model
n_p	Total number of parameters in a model
n_a	Total number of actions (primitive tasks affecting world relations) in a model
n_f	Total number of facts in an input state
n_s	Total number of steps in a plan generated by an external planner
n_n	Total number of failure annotations for a trace
m_p	Maximum number of parameters for a single task
n_b	Total number of elements of any sort in feedback provided for a trace
m_m	Maximum number of methods for a single task
m_l	Maximum number of logical symbols in all slots of any task
m_r	Maximum number of transitions leading out of any reasoning state in a model
m_f	Maximum number of instances in any fact in an input state
m_c	Maximum number of concepts that any instance is a member of
C_e	Total cost of executing of primitive tasks during a particular agent execution
C_l	Total cost of checking logical expressions during a particular agent execution
C_p	Total cost of a call to an external generative planner
C_r	Total cost of mapping relations during an adaptation process

Table 37: Terms over which the analysis of complexity is performed. Lower case terms represent discrete quantities. Upper case terms represent abstract costs (which may be a function of these or other quantities). The first two terms listed are particularly important for analysis.

binding (assumed $O(1)$); thus this step is $\boldsymbol{O(m_p)}$.[5] Next there is a check that the task given state holds; the cost of this step is included in C_l.

After that check, if the task is primitive, it is executed; the cost of this execution is included in C_e. If the task is not primitive then a method is selected. There are at most m_m potential methods. For each of these methods, the provided clause needs to be checked (which is included in C_l) and then a selection occurs. Recall that the selection is assumed to be linear so the total costs outside of C_l for that step are $\boldsymbol{O(m_m)}$. Then there is a check to see if a method was found; this step is $\boldsymbol{O(1)}$ since it simply involves seeing if there is a value for a local variable. After that, the method is executed (see the analysis below). Next a value is set in the trace (assumed $\boldsymbol{O(1)}$). Then the bindings for the parameters are updated; this involves at most m_p updates each of which are assumed $O(1)$ so this step is $\boldsymbol{O(m_p)}$. Then there is a check that the makes condition holds (included in C_l). Finally, the trace is returned (assumed $\boldsymbol{O(1)}$). Since this algorithm is executed once for each invocation of a task, the total extra cost per task invocation during execution for running this algorithm is $O(1) + O(m_p) + O(m_m) + O(1) + O(1) + O(m_p) + O(1)$; removing asymptotically non-contributory terms, this total comes to $\boldsymbol{O(m_p + m_m)}$.

The method execution algorithm appears in Table 39 (and also in Section 5.1). It is possible to also analyze method execution in terms of the number of task invocations; specifically, each task invocation directly leads to at most one method invocation. The first few steps in the method execution process occur only once. Thus it is possible to assign to the cost of each of these steps to the task that invoked the method. Furthermore, there is a single loop in this algorithm and each iteration of that loop always invokes exactly one task; thus the cost of each step in the loop can be assigned to the task which is invoked in that iteration. Thus these two sets of steps are analyzed separately below and the totals for each are added to the total cost per task invocation.

The steps outside of the loop in method execution are very simple. They involve two trace element creations, one slot value access, three slot value assignments, and one return statement at the end. Since all of these kinds of steps are assumed $O(1)$, these steps add a total of $\boldsymbol{O(1)}$ to the cost per task.

[5]Note that this and the other checks described here also pose the possibility that the check fails. If this happens, a failure annotation is created and its slots are set to the appropriate values. This process involves only operations assumed to be $O(1)$. Thus the possibility of failed checks does not affect the asymptotic analysis of the costs.

Algorithm **execute-task**(main-task)	
Inputs:	main-task: A task to be executed
Outputs:	main-task-trace: A task-trace for the execution
Other Knowledge:	current-knowledge-state: The knowledge used and produced
Effects:	The task has been executed.

main-task-trace = NEW task-trace
main-task-trace:task = main-task
main-task-trace:input-knowledge-state = current-knowledge-state
CHECK [that the parameters in main-task:input are bound in current-knowledge-state]
CHECK [that main-task:given holds]
IF [main-task is primitive] THEN
  [do main-task]
ELSE
  chosen-method = DECIDE [on an applicable method for main-task]
  CHECK [that a method was found]
  chosen-method-trace = **execute-method**(chosen-method)
  main-task-trace:invoked-mmethod-trace = chosen-method-trace
[update bindings in current-knowledge-state]
main-task-trace:output-knowledge-state = current-knowledge-state
CHECK [that main-task:makes holds]
RETURN main-task-trace

Table 38: Algorithm for task execution

Algorithm **execute-method**(chosen-method)	
Inputs:	chosen-method: A method to be executed
Outputs:	chosen-method-trace: A method-trace for the execution
Other Knowledge:	current-knowledge-state: The knowledge used and produced
Effects:	The method has been executed.

chosen-method-trace = NEW method-trace
chosen-method-trace:mmethod = chosen-method
current-transition = chosen-method:start
current-transition-trace = NEW transition-trace
current-transition-trace:transition = current-transition
chosen-method-trace:start-trace = current-transition-trace
WHILE [current-transition is not terminal]
  current-reasoning-state = current-transition:next
  current-reasoning-state-trace = NEW reasoning-state-trace
  current-transition-trace:next-reasoning-state-trace = current-reasoning-state-trace
  current-subtask-trace = **execute-task**(current-reasoning-state:subtask)
  current-reasoning-state-trace:subtask-trace = current-subtask-trace
  current-transition = DECIDE [on an applicable transition for current-reasoning-state]
  CHECK [that a transition was found]
  current-transition-trace = NEW transition-trace
  current-transition-trace:transition = current-transition
  current-reasoning-state-trace:next-transition-trace = current-transition-trace
RETURN chosen-method-trace

Table 39: Algorithm for method execution

The beginning of the loop is the check to see if the transition is terminal; this simply involves accessing the next slot and seeing if it has a value (assumed $\boldsymbol{O(1)}$). The next three steps involve a slot value access, a trace element creation, and a slot value assignment (all assumed $\boldsymbol{O(1)}$). Next the subtask is executed (see above). There is another $\boldsymbol{O(1)}$ slot assignment and then a next transition is selected. There are at most m_r potential transitions. For each of these transitions, the provided clause needs to be checked (which in included in C_l) and then a selection occurs. Recall that the selection is assumed to be linear so the total costs outside of C_l for that step are $\boldsymbol{O(m_r)}$. There is a $\boldsymbol{O(1)}$ check that a transition was, in fact, found. Finally, there are some more $\boldsymbol{O(1)}$ trace manipulations. Thus the total extra cost per task invocation for the method execution process is $\boldsymbol{O(m_r)}$.

Combining the costs from the two algorithms, the total cost outside of primitive execution and logical assertion checking for each time that a task is invoked is $O(m_p + m_m + m_r)$. Since each task invocation corresponds to a task trace in the trace, the total number of task invocations must be no more than t. Thus total costs of execution of a TMKL model is $\boldsymbol{C_e + C_l + O(t \cdot (m_p + m_m + m_r))}$. The parameters, methods, and reasoning states are all part of the model so the counts of specific quantities of these items must be less than the total number of elements in the model, i.e., $m_p + m_m + m_r \leq s$. In fact, one would expect that in practice that any given task or reasoning state in a model would have a relatively small number of parameters, methods, or transitions. If one can assume that these values are all bound by some fixed constant, then the total cost of execution becomes $C_e + C_l + O(t)$. However, in the more general case, the total cost is $C_e + C_l + O(s \cdot t)$. In other words, the extra costs imposed by the execution algorithm are typically linear in the total size of the trace produced during the execution, but may, for extremely intricate models, be proportional to the *product* of the size of the trace and the size of the model.

7.2.2 Complexity of Adaptation

This subsection presents the complexity of the major adaptation algorithms from Chapter 6. As in the previous section, total costs for individual steps and entire algorithms are indicated in boldface.

7.2.2.1 Generative Planning Adaptation

Table 40, which also appears in Section 6.2, presents the algorithm for the adaptation mechanism which uses generative planning. The uncontrollable cost for this mechanism is C_p, the cost incurred by the external planning system. In general, this cost can be very large; further analysis depends on the details of the external planning system used. This section focuses on the additional costs beyond C_p which are imposed by the adaptation mechanism.

The first step in the algorithm, translation of actions, involves first retrieving all of these actions; there are n_a such actions and the retrieval is assumed to be linear so this portion of the step is $O(n_a)$. The translation involves converting the logical assertions in the preconditions and postconditions of the primitive action to assertions in the external planners language. This is a direct one-to-one syntactic translation; the total cost is proportional to the number of symbols being translated, which is at most m_l per action, for a total of $n_a \cdot m_l$. Thus the total cost for this step is $O(n_a) + O(n_a \cdot m_l)$ which reduces to $\boldsymbol{O(n_a \cdot m_l)}$.

The second step in the algorithm finds all of the relevant relations in slots in one of the primitive tasks or the main task. This involves accessing $n_a + 1$ tasks; for each task, the maximum number of relations which could be present is m_l, and for each relation, the cost of accessing that relation assumed to be $O(1)$. Thus the cost of this step is $O((n_a + 1) \cdot m_l)$ which reduces to $\boldsymbol{O(n_a \cdot m_l)}$.

The next step is another translation process which again costs $\boldsymbol{O(m_l)}$. The following step involves accessing items in the atomic facts in the input knowledge state and in the goal (translated in the previous step). There are n_f such facts, each of which has at most m_f references. The goal has at most m_l references. For each reference, there is an $O(1)$ access, so the total cost for the step is $\boldsymbol{O(m_l + n_f \cdot m_f)}$. After that, the types of the objects are accessed; recall that this is assumed to be linear in the number of types so each of the objects requires $O(m_c)$, for a total cost of $\boldsymbol{O(m_c \cdot m_l + m_c \cdot n_f \cdot m_f)}$.

After that, all of the accumulated pieces of information are sent to the external planning system, incurring the C_p cost. Once the planning mechanism is done, a new method is created (which is assumed $\boldsymbol{O(1)}$). Next the subtasks for that new method are extracted; this involves $O(1)$ time for each of the n_s steps in the plan for a total of $\boldsymbol{O(n_s)}$. Then the transitions are produced to connect the subtasks. This involves moving through the n_s steps in the plan. At each step the transition needs to be created, which is assumed $O(1)$ and bindings for the parameters need to be created. There are at most m_p parameters and the bindings are assumed to take $O(1)$ to create so this step has a total cost of $\boldsymbol{O(n_s \cdot m_p)}$. Finally the new method is added to the task and the task is returned; these steps are assumed $\boldsymbol{O(1)}$.

Algorithm **adapt-using-generative-planning**(main-task)	
Inputs:	main-task: The task to be adapted
Outputs:	main-task: The same task with a method added
Other Knowledge:	current-knowledge-state: The knowledge to be used by the task
	primitive-actions: A set of primitive actions which can be performed
Effects:	A new method has been created which accomplishes main-task within current-knowledge-state using the primitive-actions.

operators = [translate primitive-actions]
relevant-relations = [all relations in any slot of main-task or any of primitive-actions]
assertions = [translate all facts in current-knowledge-state involving any relevant-relations]
goal = [translate main-task:makes]
objects = [names of all objects which are directly referenced in facts or goal]
object-types = [the concepts for which the objects are instances]

plan = [invoke planner on operators, assertions, objects, object-types, and goal]

new-method = NEW method
[subtasks for new-method] = [extract actions from plan]
[transitions for new-method] = [infer transitions from plan]
[add new-method to main-task:by-mmethod]
RETURN main-task

Table 40: Algorithm for adaptation by generative planning

Adding up the various steps, the total cost of the algorithm comes to $C_p + O(n_a \cdot m_l) + O(n_a \cdot m_l) + O(m_l) + O(1) + O(m_l + n_f \cdot m_f) + O(m_c \cdot m_l + m_c \cdot n_f \cdot m_f) + O(1) + O(n_s) + O(n_s \cdot m_p) + O(1)$. Removing non-contributory terms, this equates to $\boldsymbol{C_p + O(n_a \cdot m_l + m_c \cdot m_l + m_c \cdot n_f \cdot m_f + n_s \cdot m_p)}$. In practice, the following terms are typically very small: m_l, m_c, m_f, m_p. If these values can be bounded by a constant then the total cost of the algorithm reduces to $C_p + O(n_a + n_f + n_s)$. Thus the total cost of execution of the generative planning strategy is equal to the total cost incurred by the external planner plus additional processing which is typically linear in the number of actions, the number of facts in the input, and the length of the plan produced by the planner. Furthermore, it is reasonable to expect that the planner probably accesses all of the input actions and facts at least once and it certainly accesses all of the steps in the plan it produces at least once. Thus C_p is almost certainly likely to be $\Omega(n_a + n_f + n_s)$; i.e., it asymptotically dominates the additional cost imposed by the other steps of the algorithm. Thus under the stated assumptions, it can be concluded that the generative planning mechanism does not add any asymptotic cost above the cost of running the external planner.

7.2.2.2 Relation Mapping

Table 41, which also appears in Section 6.3, presents the algorithm for relation mapping. Like the previous strategy, there is an arbitrarily large fixed cost to the strategy: the total cost of trying relations to see if they map the costs of the known task and the new task. This cost is labeled C_r. If all of the relations within an agent are implemented by look-up tables, then the mapping process will be a simple search through the (at most) m_l terms looking up items. In this case, C_r will be $O(m_l)$. However, some formalisms (including Loom and thus TMKL) also allow other types of specifications such as arbitrary program code which determines if the relation holds. The running time for that arbitrary relation is obviously unlimited.

The first step involves copying an existing method (including the methods' subtasks and those subtasks' methods). This process includes copying each of the individual steps of the method. The copying moves through each piece of the model below the method (at most s elements). At each piece a new object is created, the old object's slots are accessed, and the new object's slots are set. Each of these operations is $O(1)$ and the maximum number of slots in each object is fixed (by the TMKL ontology). Thus the combined cost of the first step is $\boldsymbol{O(s)}$.

Algorithm **relation-mapping**(main-task, known-task)	
Inputs:	main-task: The task to be adapted
	known-task: A similar task with at least one method
Outputs:	main-task: The original task with a method added
Other Knowledge:	current-knowledge-state: The knowledge to be used by the task
Effects:	A new method has been created which accomplishes main-task within current-knowledge-state using a modified version of a method for known-task.

```
main-task:by-mmethod = COPY known-task:by-mmethod
map-relation = [relation which connects results of known-task to results of main-task]
mappable-relations = [all relations for which map-relation holds with some relation]
mappable-concepts = [all concepts for which map-relation holds with some concept]
relevant-relations = mappable-relations + [all relations over mappable-concepts]
relevant-manipulable-relations = [relevant-relations which are internal state relations]
candidate-tasks = [all tasks which affect relevant-manipulable-relations]
FOR candidate-task IN candidate-tasks DO
      IF [candidate-task directly asserts a relevant-manipulable-relations] THEN
         [impose the map-relation on the assertion for that candidate task]
      ELSE IF [candidate-task has mappable output] THEN
         [insert an optional mapping task after candidate-task]
RETURN main-task
```

Table 41: Algorithm for relation mapping

The second step involves searching for a mapping relation; the total cost for this step is defined to be C_r, as described above. The next two steps both involve determining if this relation holds. Again, this may be arbitrarily expensive. For the purpose of this analysis, assume that the chosen relation can be evaluated in $O(1)$ time; i.e., while there may be expensive relations in the system (whose costs are included in C_r), the one relation which is actually found is cheap to use. If that is not true, then the next few steps may be very expensive as well. The next step involves finding pairs of relations which the map relation connects. There are n_r^2 pairs of relations so this process is $\boldsymbol{O(n_r^2)}$ to check each pair individually (since checking if a relation holds is assumed $O(1)$). This bound may not be tight if the internal representation of the relation enables a more efficient access to that information. The search for pairs of concepts that are mapped by this relation is similarly $\boldsymbol{O(n_c^2)}$. The next step finds relations that affect mappable concepts; this involves going through the mappable relations (of which there can be no more than n_r) and extracting the concepts which those relations involve. Recall that extracting concepts from a relation is assumed to be linear in the number of concepts extracted; there can be no more than n_c concepts in each relation so the cost for each relation must be $O(n_c)$ bringing the total cost for the step to $\boldsymbol{O(n_r \cdot n_c)}$.

The next step involves filtering out relations which are not internal state relations; in REM this characteristic is represented directly so each relation can be checked in constant time, making the step $\boldsymbol{O(n_r)}$. The next step involves stepping through all of the tasks in the agent (of which there are n_t) and checking the terms in the slots of that task (of which there are at most m_l) to see if they match any of the relevant manipulable relations (of which there are at most n_r); thus this step is $\boldsymbol{O(n_t \cdot n_r \cdot m_l)}$.

After that step is a loop. The loop executes for each candidate task (thus at most n_t times). The first step of the task involves looking through the asserts slot of the task (at most m_l terms) and finding if there is an item which matches the (at most n_r) relevant manipulable relations; thus this step is $\boldsymbol{O(n_r \cdot m_l)}$. If there is a match, the relation is inserted into the asserts slot (assumed $\boldsymbol{O(1)}$). Otherwise, there is a check to see if the task has a mappable output. This involves searching through the (at most m_p) output parameters of the task and checking if the concept for the parameter (which is assumed to be accessible in $O(1)$ time) is involved in the map relation; recall that accessing the concept involved in a parameter is $\boldsymbol{O(n_c)}$. Thus the total cost of that check is $\boldsymbol{O(n_c \cdot m_p)}$. Finally, if that check is met then an optional insertion task is created; this involves a fixed number of instance creations and slot value settings and is thus assumed $\boldsymbol{O(1)}$. Consequently, the total cost for each iteration in the loop is $O(n_r \cdot m_l + n_c \cdot m_p)$.

Algorithm **failure-driven-model-transfer**(main-task, trace, feedback)	
Inputs:	main-task: The task to be adapted
	trace: A trace of the execution of main-task
	feedback: Additional information, if any, provided by the user regarding the execution
Outputs:	main-task: The original task with some modification
Other Knowledge:	*none*
Effects:	Some portion of the main-task and/or its methods, subtasks, etc. has been modified to address a failure of the trace

```
analyze-feedback(main-task, trace, feedback)
REPEAT
    failure-annotation = select-failure(trace)
    [apply relevant repair mechanism if any for failure-annotation]
UNTIL [repair is successful] OR [there are no more failures]
RETURN main-task
```

Table 42: Main algorithm for failure-driven model transfer

Since there at most n_t iterations, the total cost of the loop is $\boldsymbol{O(n_t \cdot n_r \cdot m_l + n_t \cdot n_c \cdot m_p)}$.

Adding the cost of the earlier steps to the cost of the loop produces a total of $O(s)+O(n_r^2)+O(n_c^2)+O(n_r \cdot n_c)+O(n_t \cdot n_r \cdot m_l)+O(n_t \cdot n_r \cdot m_l)+O(n_t \cdot n_r \cdot m_l + n_t \cdot n_c \cdot m_p)$. Combining terms, this comes to $\boldsymbol{O(s + n_r^2 + n_c^2 + n_r \cdot n_c + n_t \cdot n_r \cdot m_l + n_t \cdot n_c \cdot m_p)}$. Note that n_t, n_r, and n_c (the numbers of tasks, relations, and concepts) are all parts of s (the total size of the model). Furthermore, in a typical model each of n_t, n_r, and n_c are likely to represent a significant fraction of the model, so it is reasonable to consider the analysis in the context of the fact that all these are bounded by s. This fact leads to the claim that the total cost is $O(s + s^2 + s^2 + s^2 \cdot m_l + s^2 \cdot m_p)$ which equates to $O(s^2 \cdot m_l + s^2 \cdot m_p)$. Recall from earlier analysis that m_p and m_l are typically small and may often be bounded by a constant. In that case, the total cost of the algorithm is $O(s^2)$. More generally, m_p and m_l are bounded by s, so the total cost of the algorithm is $O(s^3)$. Thus the cost of relation mapping is typically quadratic in the size of the model but may be cubic in the size of the model for particularly intricate models.

7.2.2.3 Failure-Driven Model Transfer / Fixed Value Production

Table 42, which also appears in Section 6.4, presents the general algorithm for failure-driven transfer. The first step in the process, feedback analysis, is considered later in this section. After that step, there is a loop which executes until one of the failures has been successfully addressed or all of the failures have been attempted. There are n_n failures, so the loop executes at most n_n times.

The first step in the loop, failure selection, involves stepping through the failure annotations, assigning them a priority, and then choosing the one with the highest priority that has not yet been attempted. The assignment of priorities is done using some fixed heuristics which merely access the failure type and the type of model element which the trace records, so this portion of the process is $O(1)$ for each failure, making the assignment process $O(n_n)$. The selection of the largest is done by a traditional linear search which is also $O(n_n)$. Hence the combined cost for failure selection is $O(n_n)$ per iteration. Since there are at most n_n iterations, the total cost of failure selection in the entire adaptation process is $O(n_n^2)$.[6]

The next step in the loop involves selecting and applying a repair mechanism. Selecting is done by heuristics which merely access and compare slot values, so that process is $O(1)$. In many cases, a failure will have no valid repair mechanism (the class of failures which can be represented using the notations in Section 3.3.2 is much larger than the class of failures which the existing repair mechanisms can address). In this case, the loop continues until if

[6] As an aside, it would be possible to prioritize and sort the failures outside of the loop. Since there are only a fixed number of possible priorities, this could be done by a simple table sort in $O(n_n)$ time and then failure selection within the loop would just step through the list in $O(1)$ time per iteration. This would bring the total time for failure selection within the algorithm from $O(n_n^2)$ to $O(n_n)$. This has not been done in REM because n_n tends to be very small in practice making this potential optimization fairly insignificant.

Algorithm **analyze-feedback**(main-task, trace, feedback)	
Inputs:	main-task: The task to be adapted
	trace: A trace of the execution of main-task
	feedback: Additional information, if any, provided by the user regarding the execution
Outputs:	trace: The original trace, possibly with some additional annotations
Other Knowledge:	*none*
Effects:	Conflicts between what the trace indicates happened and what the feedback indicates should have happened are identified as failure annotations on the trace.

```
FOR feedback-term IN feedback DO
    FOR trace-element IN [search trace for potential contradictions to feedback-term]
        IF [feedback-term directly contradicts trace-element] THEN
            failure-annotation = NEW directly-contradicts-feedback annotation
        ELSE
            failure-annotation = NEW may-contradict-feedback failure annotation
        failure-annotation:desired-input = [input state of feedback]
        failure-annotation:desired-output = [output state of feedback]
        failure-annotation:actual-input = [input state of trace]
        failure-annotation:actual-output = [output state of trace]
        [attach failure-annotation to trace-element]
RETURN trace
```

Table 43: Algorithm for feedback analysis

finds a failure which it can address. The two mechanisms which it may eventually settle on are generative planning and fixed value repair. The cost of repair depends on which of these mechanisms is used; the costs for generative planning appears in Section 7.2.2.1 and the costs for fixed value repair are described later in this section.

Table 43 presents the feedback analysis algorithm which also appears in Section 6.4. The outer loop in the algorithm steps through the (at most n_b) terms in the feedback provided. The inner loop searches through the (at most t) elements of the trace looking for potential contradictions. Each check for a contradiction involves comparing parameters and values from the feedback to those in the trace. The total number of parameter bindings at a given step is bounded by the total number of parameters in the model (n_p). The total number of parameter bindings in the feedback is bounded by both the number of parameters in the model and the number of elements in the feedback (n_p and n_b). Each check to see if two bindings match involves only slot accesses and value comparisons and is thus $O(1)$. Hence the combined cost of a check to see if a trace element contradicts feedback is bounded by both $O(n_p^2)$ and $O(n_p \cdot n_b)$. In practice, the amount of feedback provided by the user is likely to be fairly small so $\boldsymbol{O(n_p \cdot n_b)}$ is likely to be a tighter bound (and thus it is used for the totals generated later). Within the loop there is a check to see if the contradiction is a direct one; this check involves accessing the task referred to in the trace element and the one referred to in the feedback element and seeing if they match, which demands a fixed number of basic $O(1)$ operations. The remaining items within the loop all involve simply creating instances plus accessing and setting slots. Thus the total cost of the computation in the interior loop is $\boldsymbol{O(n_p \cdot n_b)}$. Since the bounds on the number of iterations of the inner and outer loop are t and n_b, the total cost of the loop is $O(t \cdot n_p \cdot n_b^2)$. The last step of the algorithm returns a value; this step is assumed $O(1)$. Thus the total cost of the feedback analysis algorithm is $\boldsymbol{O(t \cdot n_p \cdot n_b^2)}$.

Table 44 presents the fixed value production repair strategy from Section 6.4.2. The first five steps all simply involve slot accessing, slot setting, and instance creation; these are all assumed to be $O(1)$. After those steps are two loops. These loops step through different pieces of the desired inputs and outputs. These desired states were specified in the feedback so the total number of items in them are bounded by n_b. The steps in the loop involve adding values to a slot which is assumed $O(1)$ so the total cost of the loop is $\boldsymbol{O(n_b)}$. The remaining steps involve a fixed number of instance creations plus a fixed number of slot value access, set, add, and delete operations; all of these computations are assumed $O(1)$. Thus the total cost of the fixed value production repair algorithm is $\boldsymbol{O(n_b)}$.

Algorithm **fixed-value-production-repair**(failure-annotation)	
Inputs:	failure-annotation: a record of a task failure
Outputs:	*none*
Other Knowledge:	task-trace: The trace to which failure-annotation is attached
	original-task: The task for which task-trace records the execution
Effects:	An alternative method has been created for original-task which produces results indicated by failure-annotation.

original-task = [the task executed in the trace element for failure-annotation]
base-task = [copy of original-task]
alternative-task = NEW task
alternative-task:input = original-task:input
alternative-task:output = original-task:output
FOR binding IN failure-annotation:desired-output
    [add binding to the alternative-task:binds]
FOR binding IN failure-annotation:desired-input
    [add a check that binding holds to alternative-task:given]
base-method = NEW method
[create state machine for base-method that simply invokes base-task once]
alternative-method = NEW method
[create state machine for alternative-method that simply invokes alternative-task once]
alternative-method:provided = alternative-task:given
[delete implementation of original-task]
original-task:by-mmethod = [list of base-method and alternative-method]

Table 44: Algorithm for fixed value production repair

Note, also, that there are no possible failure points in the algorithm.[7] Thus whenever it is selected, the repair is considered successful and the main loop for the failure driven transfer process (in Table 42) terminates; thus it is guaranteed that the fixed value repair mechanism will only be executed once for a given adaptation.

Returning to the analysis of the main failure-driven transfer algorithm, it is now possible to combine the costs of the different pieces. The feedback analysis costs $\boldsymbol{O(t \cdot n_p \cdot n_b^2)}$. Failure selection costs $\boldsymbol{O(n_n^2)}$. Fixed value production repair occurs at most once and costs $\boldsymbol{O(n_b)}$. Thus the combined cost of failure-driven transfer using fixed value production repair is $O(t \cdot n_p \cdot n_b^2) + O(n_n^2) + O(n_b)$ which is equivalent to $\boldsymbol{O(t \cdot n_p \cdot n_b^2 + n_n^2)}$. In practice, the number of parameters in a model (n_p) is typically fairly small (for example, there are 27 parameters in ADDAM and 8 in the web browsing agent); thus it may be reasonable to treat this value as constant, but more generally it is bounded by s (because s is the size of the model and the parameters are part of the model). The size of the feedback n_b is likely to be fairly small in practice (because a user is not going to want to provide an enormous amount of information), but there is no theoretical bound on that size. Regarding the number of failures, n_n, note that there are a fixed number of ways that a failure can occur at each step in a process; thus the number of failure annotations is bounded by some constant times the number of steps in the trace (t); in practice, the number of failed steps in the trace are likely to be a very small fraction of the total size of the trace so this quantity may also be reasonably considered to be bounded by a constant. This leads to several different analyses depending on assumptions:

- If the number of parameters, the number of failures, and the size of the feedback are all assumed to be bounded by a constant, k, then the total cost is thus $O(t \cdot k \cdot k^2 + t \cdot k)$ which is equivalent to $O(t)$.
- If the assumption about the number of failures is removed, then the cost is instead $O(t \cdot k \cdot k^2 + t^2)$ which is equivalent to $O(t^2)$.
- If the assumption about the number of parameters is then removed, the cost becomes $O(t \cdot s + t^2)$.

[7] Assuming that the input is syntactically valid to begin with. That assumption is implicit in many places throughout this dissertation; see Section 9.2 for more on this topic.

- Finally, if none of these values are considered fixed then the cost is $O(t \cdot s \cdot n_b^2 + t^2)$.

In summary, the total cost of failure-driven transfer using fixed value production repair is typically linear in the size of the trace which was generated but may also be affected by the size of the model, the amount of feedback provided, and the number of failures which were detected.

7.2.2.4 Failure-Driven Model Transfer / Generative Planning

Recall that their are two different failure-driven model transfer mechanisms presented in Chapter 6: fixed value production and generative planning. The latter is also used in isolation as an adaptation strategy for a new task. The analysis of the generative planning strategy appears in Section 7.2.2.1. The analysis of failure-driven transfer using fixed value production appears in Section 7.2.2.3. The analysis of failure-driven transfer using generative planning is very simple given these previous analyses. The fixed value repair mechanism did not add to the asymptotic costs of the overall process using that mechanism. Since the other portions of the process (feedback analysis and failure selection) are also used in transfer by generative planning repair, the total asymptotic cost computed in Section 7.2.2.3 is one portion of the total cost of transfer by planning.

The other portion of the process is simply the generative planning mechanism. As stated in Section 7.2.2.1, the cost imposed by that mechanism is asymptotically dominated by the cost of the generative planning system. Thus the total combined worst case cost for failure-driven model transfer using generative planning is equal to the cost of the external planning process plus an additional cost which is typically linear in the size of the trace which was generated but may also be affected by the size of the model, the amount of feedback provided, and the number of failures which were detected; these additional effects are described more precisely in Section 7.2.2.3.

7.2.3 Analysis Summary

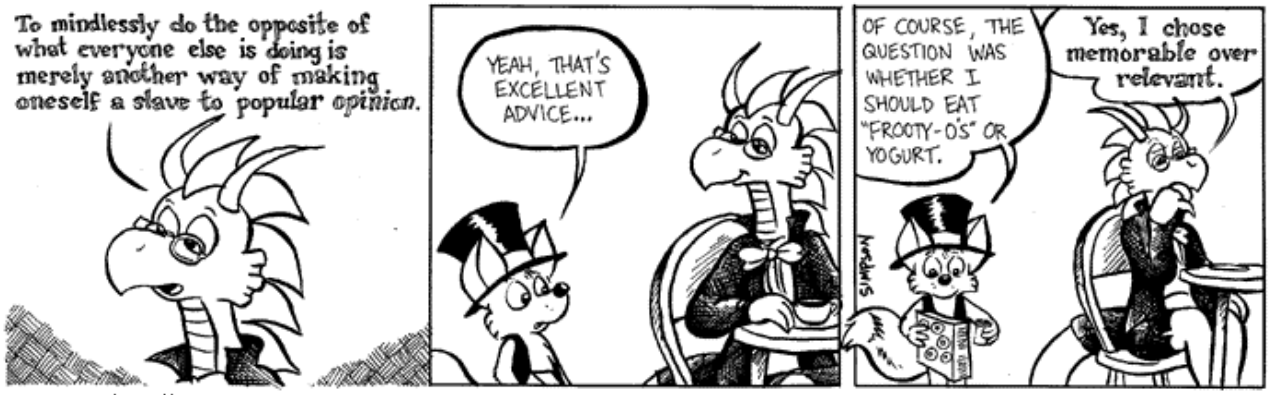
The analyses in the previous subsections hinge on a variety of assumptions stated earlier in the section. Under these assumptions, the following conclusions were drawn:

- The cost of execution is potentially unlimited because an arbitrary agent may be arbitrarily expensive to run. It is possible, however, to analyze the extra costs imposed by the execution algorithm on top of the total costs of executing the pieces of the process separately. The analysis shows that these extra costs are typically linear in the total size of the trace produced during the execution, but may, for particularly intricate models, be proportional to the product of the size of the trace and the size of the model.
- The generative planning adaptation mechanism invokes an external planning system which may be very expensive. The mechanism does not add any asymptotic cost above the cost of running the external planner.
- The cost of the relation mapping mechanism is typically quadratic in the size of the model but may be cubic in the size of the model for particularly intricate models.
- The cost of failure-driven transfer using fixed value production repair is typically linear in the size of the trace which was generated but may also be affected by the size of the model, the amount of feedback provided, and the number of failures which were detected.

The combination of these analytical results and the experimental results presented in Section 7.1 forms the evaluation for the work presented in this dissertation. This evaluation verifies that the representations and reasoning processes presented in earlier chapters are able to effectively address a variety of problems including some problems of reasonably large scale. This fact forms the basis for the discussion which occurs in the following chapters.

Chapter 8

Related Research



There is a very large body of research which has some relationship with the work described in this dissertation. Three broad areas which are particularly relevant to this work are Artificial Intelligence, Cognitive Science, and Software Engineering. The first section of this chapter provides an overview of related research in Artificial Intelligence. The next section provides a more specific view of the relationship that this work has to past work involving TMK models. The third section presents a brief look at the relationship between this work and the field of Cognitive Science. Finally, the fourth section describes the relationship that this work has to the field of Software Engineering. This chapter provides a context for understanding how the ideas and results presented throughout this dissertation build upon and contribute to existing research.

8.1 Artificial Intelligence

Intelligent agents typically have a substantial amount of information. Some of this information is explicitly represented in a general-purpose formalism which can support a wide variety of inferences; information of this sort is generally referred to as "knowledge." Other information in a system may only be present implicitly in the form of specialized algorithms and data structures. A key claim of almost all research in AI is that characteristics of an agent (such as efficiency, flexibility, etc.) are often heavily affected by which portions of the agent's information are explicitly encoded as knowledge and how they are encoded.

One obvious technique for creating software which can perform some activity is to simply write a special-purpose program which simply does that and nothing else. All the information about objects, actions, processes, etc. certainly has to be in this software, but most of that information is represented implicitly. Such a system can be very fast because its processing is specifically tailored to the purpose that it is designed for. The effectiveness of this approach is evident in the overwhelming commercial success that it has had. Software stores are filled with productivity tools, utilities, networking software, games, etc. Virtually one hundred percent of these are built by human programmers for some specific purpose.

There are, however, disadvantages to building systems by hand for a specific purpose. The most striking drawback of a hard-coded system is that it is not able to adapt to new challenges. If the system encounters a situation or a

	Generative Planning	Model-Based Adaptation
Why does deliberation occur?	To develop a new sequence of actions	To alter the effects of an existing reasoning strategy
What is being deliberated on?	Goals and actions	Functional models of reasoning strategies
How does deliberation occur?	Deduction and/or search	A variety of model-based strategies
When does deliberation occur?	Before any action	Before or after a reasoning process
Summary	Unlike model-based adaptation, generative planning does not use any knowledge of existing reasoning strategies. Thus it needs to construct an entire process from scratch every time it runs. This can be very expensive.	

Table 45: Comparison between generative planning and pure model-based adaptation. Note that the research presented in this dissertation contains elements of both of these but emphasizes the latter more than the former.

request that it was not programmed to address, it simply fails. One major goal of the field of Artificial Intelligence is to avoid this limitation, i.e., to build systems which are capable of addressing challenges for which they were not specifically constructed to handle. Such systems typically involve explicit representation of at least some of the sorts of information which would be implicit in a hard-coded program; the system can use this explicit information to address a problem in a more general, flexible way and thus react to a broader variety of problems. All of the Artificial Intelligence techniques discussed in the subsections below have this characteristic. The model-based reflection approach described in this dissertation uses a particular set of explicit representations to enable certain kinds of reasoning. Because there are a range of interesting and important problems which are effectively addressed by the model-based reflection work described in this dissertation and are not effectively addressed by the related approaches described below, this work constitutes an important contribution to the field as whole.

8.1.1 Generative Planning

Generative planning [Tate et al., 1990] involves constructing an entire sequence of actions which lead from the current state to the goal state. Relatively recent advances in generative planning, e.g. [Blum and Furst, 1997], have provided systems which are much faster than earlier techniques. However, the problem of assembling an entire plan for a complex goal given only descriptions of actions is fundamentally very hard, and thus even modern techniques can require an enormous amount of computation for complex goals. Furthermore, traditional generative planning systems make no use of past experience; when they encounter goals which are similar (or even identical to) those which they have addressed in the past, they simply construct a new plan for those goals from scratch. Thus the enormous cost of constructing a plan must be incurred over and over.

Table 45 provides a comparison between generative planning and pure model-based adaptation. The two approaches are similar in that they both involve explicitly deliberating about what actions to perform. However, model-based adaptation involves reusing existing processes and the processes that it reuses are encoded as functional models of reasoning strategies rather than simple combinations of actions. The reuse capabilities of model-based adaptation can be a substantial advantage in efficiency (as shown in Chapter 7). However, pure model-based adaptation has a significant drawback in that it is helpless if there is no relevant reasoning process or if even one part of the existing reasoning process is completely unsuited to some new demands. Consequently, REM combines these two approaches; existing processes are reused through model-based adaptation when possible, and when a process or some portion of a process cannot be reused, then generative planning and/or reinforcement learning are employed.

One major difference between TMKL models and many kinds of plans is that the former are hierarchical while the latter are merely a flat sequence. However, it is possible to build hierarchical plans; in these plans high-level actions are decomposed into more detailed low-level actions. Hierarchies of actions provide explicit advance knowledge about how the low-level actions combine to accomplish particular subgoals [Sacerdoti, 1974]. Like TMKL models, hierarchies of actions are an additional knowledge requirement which can enable avoidance of the potentially explosive cost of sifting through many different possible combinations of actions. It is possible to avoid this knowledge requirement for hierarchical planning by automatically learning hierarchies, e.g., [Knoblock, 1994]. This approach sacrifices some

	Case-Based Planning	Model-Based Adaptation
Why does reuse occur?	To provide a plan for a goal which resembles or matches one in case memory	To provide a method for a task which resembles or matches one in a model
What is being reused?	Sequences or hierarchies of actions	Reasoning processes
How does reuse occur?	Domain-specific rules	Adaptation strategies which are specific to particular kinds of differences in functionality.
When does reuse occur?	Before any action	Before or after a reasoning process
Summary	Case-Based Planning and Model-Based Adaptation serve very similar purposes: reusing an existing process to accomplish a specific desired result. However, the more abstract, elaborate nature of TMKL supports more generality for the adaptation mechanisms which operate over it.	

Table 46: Comparison between case-based planning and model-based adaptation.

of the efficiency benefits of hierarchical planning (because there is some cost to learning hierarchies) in exchange for weaker knowledge requirements.

Whether hierarchies are provided by a human or developed automatically, they do have significant differences from TMKL models. Unlike TMKL models, hierarchical plans do not represent reasoning processes. They encode actions at different levels of abstraction, but do not encode any processes which involve modifying, storing, or retrieving knowledge; some planning frameworks do include sensing actions, which produce some knowledge but this is only a small fraction of a general knowledge manipulation framework. Furthermore, actions in a hierarchical planning system are typically encoded in a very restrictive formalism because these actions need to be composed dynamically (as in traditional planning); reasoning about the composition of actions encoded in very expressive formalisms tends to be prohibitively expensive. In contrast, the effects of tasks in TMKL (the given and makes slots) are encoded as arbitrary expressions in Loom, an extremely powerful formalism. This encoding is feasible because TMKL models have methods which compose the tasks (rather than performing composition automatically); new problems are addressed by making incremental revisions to an existing process rather than building a new one. Refinement planning [Doan et al., 1995] is a framework which uses hierarchies and also contains some prior knowledge of the order of actions to take (similar to methods in TMKL) as well as utilities of actions (similar to the Q-learning values in REM). However, like other planning approaches, refinement planning focuses on actions and on a single processing strategy; in contrast, the work described here involves a variety of adaptation strategies for modifying integrated models of reasoning *and* action.

8.1.2 Case-Based and Analogical Planning

There have been numerous attempts to extend the planning approach to make use of experience. For example, case-based planning [Hammond, 1989] involves retrieving and adapting past plans to address new goals. Table 46 provides a comparison of case-based planning and model-based adaptation. Case-based planning can be very quick when there is a plan in memory which is extremely similar to one which accomplishes the current goal. The biggest difference between plans and TMKL models is that the latter involve both reasoning *and* action. Furthermore, a TMKL model represents an abstract process which is appropriate for a range of inputs while a plan represents a specific combination of actions for a particular situation. One consequence of this fact for reuse is that the kinds of changes which are made in case-based planning typically involve relatively minor tweaks, so if a plan is required which is not very close to a known plan, the case-based planner is helpless. It is possible to combine case-based and generative planning to solve different portions of a planning problem [Melis and Ullrich, 1999]. This can provide some of the efficiency benefits of case-based planning and still keep the breadth of coverage of generative planning. However, it still suffers from the cost problems of generative planning when cases are not available and the limited nature of plan adaptation when they are available. A crucial drawback that systems based on the reuse of planning results suffer from is that plans encode rather limited knowledge; they describe actions and ordering of actions, but they do not describe reasoning processes whereby these actions are selected.

Like case-based planning, the model-based adaptation techniques described in this dissertation also exploit similarity between problems. However, while case-based planners demand that similar problems lead to similar sequences (or hierarchies [Muñoz-Avila et al., 1999]) of actions, model-based adaptation instead demands that similar problems lead to similar reasoning processes (even if those reasoning processes lead to very different actions). Consider, for example, the disassembly to assembly adaptation example in Section 6.3. The assembly plans in these examples bear virtually no superficial resemblance to the original disassembly plans upon which they were based: there is typically no overlap at all in the operators used (since all of the operators are inverted) and while the objects of those operators are similar, they are manipulated in the reverse order. However, the processes by which the disassembly plans and assembly plans are produced are very similar (as evidenced by the relatively small differences between the initial and final models produced in the example). Consequently, while this problem is ill-suited to traditional case-based reasoning, it is well-suited to model-based adaptation. Similar results in the other examples presented in this dissertation show that model-based adaptation is appropriate for many problems for which similar requirements lead to radically different solutions but do so through relatively similar processes.

It is possible to learn enhancements to the planning process itself. For example, PRODIGY [Veloso et al., 1995] uses a variety of techniques to learn heuristics which guide various decision points in the planning process. Learning enhancements to planning can be used for a wide range of effects, such as improving efficiency via derivational analogy [Veloso, 1994] or improving plan quality via explanation-based learning [Pérez and Carbonell, 1994]. However, these techniques assume that there is a single underlying reasoning process (the generative planning algorithm) and focus on fine tuning that process. This is very effective for problems which are already well-suited to generative planning, making them even more well-suited to planning over time as the planner becomes tuned to the domain. However, it does not address problems which are ill-suited to planning to begin with. If a problem is prohibitively costly to solve at all by generative planning, then there is no opportunity for an agent learning about the planning process to have even a single example with which to try to improve the process.

There are other case-based reasoning approaches which explicitly reason about process. For example, case-based adaptation [Leake et al., 1995] considers the reuse of adaptation processes within case-based reasoning. Case-based adaptation does not use complex models of reasoning, because it restricts its reasoning about processes to a single portion of case-based reasoning (adaptation) and assumes a single strategy (rule-based search). This limits the applicability of the approach to the (admittedly quite large) set of problems for which adaptation by rule-based search can be effectively reused. It does provide an advantage over our approach in that it does not require a functional model and does not require any representation of the other portions of the case-based reasoning process. However, given that that agents are designed and built by humans in the first place, information about the function and composition of these agents should be available to their builders (or a separate analyst who has access to documentation describing the architecture of the agent) [Abowd et al., 1997]. Thus while our approach does impose a significant extra knowledge requirement, that requirement is evidently often attainable, at least for well-organized and well-understood agents.

8.1.3 Planning over Learning Actions

It is also possible for a planning system to encode some reasoning and/or learning information in the form of actions for a planning mechanism [Cox, 1996, Shippey et al., 1998]. For example, a disassembly agent could potentially form a plan to first determine what all of the subsystems of a device were and then to disconnect those subsystems and then to disassemble them separately. A case-based planning system could even reuse this processing information [Murdock et al., 1997]. However, a major problem that this approach faces is the fact that planning algorithms typically require that all of the effects of an action be precisely known; if the results of an action are not known, then there is no way to determine how it fits into a plan. However, it is typically not possible to completely represent the effects of actions which involve the production of new knowledge; if the agent knew what exactly what knowledge it was going to produce before it produced it, then it would already have that knowledge and thus would not need to produce it. There are planning algorithms which handle actions which produce information by allowing plan steps to depend on that information; however, because the plan still needs to account for every possible outcome, the information which can be produced is extremely limited, e.g. to a single bit [Weld et al., 1998]. Of course, any information representable in a computer can be reduced to individual bits; in practice, however, it is rather impractical to do so for complex knowledge structures.

In the research presented in this dissertation, agent models are used as knowledge for autonomous reasoning processes. This approach is able to overcome some of the assorted drawbacks of various planning approaches, which just use knowledge of actions and desirable results. In particular, because custom programmed systems are generally

very efficient, it seems desirable to have a representation which can encode processing of the sort which occurs in those systems; by explicitly modeling this processing, it becomes possible to modify that processing, thereby addressing the issue of flexibility (which is the primary drawback of hard-coded systems). Because TMK models include information about actions and desired results which is consistent with planning, as well as reinforcement learning, they can be used to support these techniques. However, because they also include additional knowledge about reasoning processes and results, they can support additional types of inferences that are specific to model-based reasoning. The combination of these pure model-based techniques with established planning and learning algorithms provides flexible, cost-effective reasoning for an assortment of problems which are not adequately addressed by any of these approaches in isolation.

8.1.4 Machine Learning

The adaptation processes described in Chapter 6 result in changes to the capabilities of the agent. Consequently, the work described in this dissertation is a form of machine learning. As such it is productive to compare this work to other relevant machine learning topics: reinforcement learning and various forms of classification learning.

Reinforcement learning [Kaelbling et al., 1996] is one popular machine learning technique. An agent operating by reinforcement learning alone simply tries out actions, observes the consequences, and eventually begins to favor those actions which tend to lead to desirable consequences. This approach requires explicit knowledge of available actions and the conditions under which those actions can be performed as well as knowledge about what sorts of results are desirable. In the disassembly domain, for example, one could build an agent which knows about actions and conditions such as: when two things are screwed together and the screw is not obstructed, it can be unscrewed. This agent would also need to be able to recognize when an object is disassembled (so it can reinforce those actions which lead to that state). Compare this with a disassembly program which is hard-coded to deal with a specific set of devices. When the hard-coded program encounters a device in its set, it can just take that device apart; however, if it encounters a device which is not in its set, it cannot do anything. In contrast, the reinforcement learning agent requires extensive trial and error, especially for complex devices, but can, in principle, handle any device which can be disassembled using the actions available. Furthermore, by changing the desired outcomes, the reinforcement learning agent could even address different challenges. For example, if an agent which had been used to do disassembly were now needed to instead assemble a device, it could just be given a new description of the desirability of the states (i.e., that states where the device is assembled are good); as long as the necessary actions were available, it would eventually learn to assemble devices. A major drawback of this approach, however, is that extensive trial and error can be extremely time consuming. Furthermore, the learning done for one particular problem is typically not at all applicable for other problems, so adapting to new challenges, while possible, is often extremely slow.

Table 47 presents a comparison between reinforcement learning and pure model-based adaptation (i.e., model-based adaptation with no reinforcement learning or generative planning). Note that there is a certain amount of synergy between the "why" portions of the comparison; many reasoning strategies involve making decisions so it seems reasonable that a technique for developing a decision policy should be useful for enhancing a reasoning strategy. The decision making mechanism in REM (in Section 5.2) describes how this can be done. Models and model-based adaptation are used to perform as much of the reasoning as they possibly can. Only when the models and adaptation strategies are not quite adequate is reinforcement learning used to fill in the gap. Thus REM mitigates the potentially explosive cost of reinforcement learning by limiting its use.

A very large portion of the field of machine learning has focussed on the task of classification, i.e., given a training set of instances with specified feature values and a new instance, infer which of the categories of instances in the training set that the new instance belongs to. A broad overview of much of this and other work appears in [Mitchell, 1997]. Table 48 describes the relationship between classification learning and the model-based adaptation performed in REM.

One approach to classification is decision tree learning [Quinlan, 1986], which builds hierarchical symbolic rules in which the features and values are arranged as nodes and links in a tree. In contrast, instance-based learning [Cover and Hart, 1967, Albert and Aha, 1991] involves computing a numerical classification based on the classifications of training instances which are "near" the current instance (where nearness is defined by a multi-dimensional space in which features are dimensions and values are positions along those dimensions). Bayesian Networks [Pearl, 1988] have characteristics of both of these approaches. Like decision trees, they build rule-like networks which describe how feature values relate to each other. However, like nearest neighbor approaches they produce results through numerical computations (in the case of Bayesian Networks, through probability equations). Of these three approaches, model-based adaptation is most similar to decision tree learning in that both operate on hierarchical

	Reinforcement Learning	Model-Based Adaptation
Why does learning occur?	To establish an optimal decision policy	To alter the effects of an existing reasoning strategy
What is being learned?	Estimated future rewards for actions	New or modified methods for tasks
How does learning occur?	Numerical formulas	Redesign of symbolic models
When does learning occur?	After each decision	Before or after a reasoning process
Summary	Reinforcement learning is very flexible but, unlike model-based adaptation, does not make any use of any advanced deliberation or existing reasoning strategies.	

Table 47: Comparison between reinforcement learning and pure model-based adaptation. Note that the research presented in this dissertation contains elements of both of these but emphasizes the latter more than the former.

	Classification Learning	Model-Based Adaptation
Why does learning occur?	To determine what class an instance belongs to	To alter the effects of an existing reasoning strategy
What is being learned?	Various sorts of rules or computations for classification	New or modified methods for tasks
How does learning occur?	A wide variety of symbolic and numerical techniques	Redesign of symbolic models
When does learning occur?	As instances are presented	Before or after a reasoning process
Summary	Classification learning involves improving a particular kind of reasoning process: selecting a class from a (usually fixed) set of classes given an instance. In contrast, model-based adaptation involves improving a wide variety of reasoning processes.	

Table 48: Comparison between classification learning and model-based adaptation.

symbolic representations. This is, however, a fairly superficial similarity. Decision trees are simply aggregations of instances and, as such, do not contain any explicit knowledge about the domain in which these instances occur or the processes which construct them. In contrast, TMKL models contain extensive knowledge about the domain and the process.

There are classification learning techniques which do make use of domain knowledge, e.g., Explanation-Based Generalization (EBG) [Mitchell et al., 1986]. This approach seems much more similar to model-based adaptation than the approaches mentioned above. In particular, both model-based adaptation and EBG use extensive existing knowledge to perform learning based on a single situation rather than aggregating information from a large quantity of isolated instances. In fact, a form of EBG is used in REM within the generative planning mechanism, specifically in generalizing a plan into a reusable TMKL method (see Section 6.2). Of course, there are major differences between REM as a whole and EBG; specifically, REM is a multi-strategy shell which uses both process and domain knowledge to learn to address new tasks while EBG uses a single strategy based only on domain knowledge to learn classification rules.

8.1.5 Meta-Knowledge

There are a variety of approaches in Artificial Intelligence to reasoning about an agent's own knowledge. A Truth Maintenance System (TMS) [McDermott and Doyle, 1980], for example, represents information not only about the current knowledge of the agent but also about which portions of the knowledge were inferred and how they were inferred. This capability is used to update inferred knowledge in response to changes in given knowledge. Loom is, among other things, a TMS [MacGregor, 1999]. Because REM is built on top of Loom, REM contains a TMS. Indeed, REM does make use of Loom's truth maintenance capabilities. For example, many adaptation strategies treat primitive tasks differently from non-primitive tasks. Primitive tasks in TMKL are not explicitly marked as such; instead, REM infers what tasks are primitive using Loom's classifier mechanism. It is possible for a primitive task to become non-primitive; in particular, if the fixed value production strategy (in Section 6.4.2) is run on a primitive task, that task is divided into two methods. The truth maintenance capabilities in Loom automatically determine that this division has made the task non-primitive. Thus if a future adaptation strategy involves that task, it will be able to correctly recognize it as non-primitive. Note, in contrast, that SIRRINE does not contain a truth-maintenance mechanism. Instead, mechanisms in SIRRINE which use inferred traits such as primitiveness of a task explicitly compute the values of those traits whenever they are needed. This can be much less efficient (since traits are computed as many times as they are needed instead of being computed once and then only updated when there is a change to the information which lead to that computation). Furthermore, it can be much less convenient to program (since the computation of traits must be explicitly coded rather than being handled automatically by a TMS). Consequently, the presence of truth maintenance capabilities is one of the significant improvements that REM contains over the earlier SIRRINE system.

Theo [Mitchell et al., 1989] also represents meta-knowledge which includes a record of how some knowledge within an agent was derived. Like Loom and other TMS frameworks, Theo contains a set of assertions plus explicit records of how inferred assertions were computed from those which were explicitly asserted to it. Unlike those frameworks, however, Theo uses a variety of learning mechanisms to speed up inference by abstracting over these records. Theo can be used by an intelligent agent to provide increasingly rapid access to inferred knowledge, as demonstrated in the CAP [Dent et al., 1992] meeting scheduling agent. CAP does not contain any knowledge of its own tasks and methods; however, it does contain the knowledge about its knowledge (encoded in Theo). It is useful to contrast the operation of CAP on top of Theo with the operation of ADDAM on top of Loom (with the latter being mediated by REM). The key knowledge items produced by CAP are predictions (used for recommendation). These predictions are produced by generic learning mechanisms built into Theo, specifically backpropogation and decision tree induction. In contrast, the key knowledge items produced by ADDAM are plans (used for simulated disassembly). These plans are constructed by ADDAM's specialized case-based hierarchical disassembly planner; Loom is used primarily to provide access and storage of the results, not to directly perform inference (although it does provide a small portion of the inferencing, via its TMS capabilities). Theo is particularly valuable for CAP because it provides the primary inference mechanism (and thus its meta-knowledge about the derivation of values is presumably complete). In contrast, the meta-knowledge in Theo would not be particularly useful for ADDAM because it mainly produces knowledge via specialized reasoning methods. Because REM explicitly models these methods (and the subtasks of which the methods are composed), it is able to reason about and adapt those methods.

One could envision a variant of REM which used Theo instead of Loom; such a variant would probably be faster for agents like CAP which rely on the underlying knowledge representation system for much of its inferencing. However,

	Meta-Planning	Model-Based Adaptation
Why does meta-reasoning occur?	To provide control between a fixed set of planning mechanisms	To alter the effects of a reasoning strategy provided by the agent's designer
What meta-knowledge is being used?	Knowledge about the domain, about a plan, and about the planning mechanisms	Knowledge about domain and reasoning mechanisms
How does meta-reasoning occur?	Planning	Redesign of models
When does meta-reasoning occur?	Interleaved with basic planning	Before or after a reasoning process
Summary	Meta-planning and model-based adaptation both make use of extensive knowledge about both a domain and a set of reasoning mechanisms. However, meta-planning draws on a fixed form of representation (plans) and a fixed set of reasoning mechanisms (such as least-commitment planning). In contrast, model-based adaptation allows an agent's designer to encode a wide range of knowledge and reasoning and then uses its own adaptation mechanisms to make adjustments as needed.	

Table 49: Comparison between meta-planning and model-based adaptation.

it seems unlikely that the meta-knowledge in Theo would provide significant benefits for an agent like ADDAM which uses specialized reasoning strategies for most of its inferencing. Furthermore, Loom certainly provides many practical benefits over Theo as a knowledge representation component since the emphasis of the Loom project has been on providing a representation tool for broad distribution to a variety of application developers. Consequently, Loom seems like a better alternative for knowledge representation in a general-purpose reflective reasoning shell like REM, unless it is known in advance that many agents encoded in the shell will be particularly well suited to Theo.

8.1.6 Meta-Reasoning

One approach to reasoning about reasoning is meta-planning. For example, MOLGEN [Stefik, 1981] performs planning and meta-planning in the domain of molecular genetics experiments. As another example, PAM [Wilensky, 1981] understands the plans of agents in stories in terms of meta-planning. Both of these systems perform planning in the context of extensive background knowledge about the domain and the available reasoning mechanisms. MOLGEN divides this background knowledge up into distinct levels and reasons about them separately while PAM uses a single integrated level for all sorts of reasoning and meta-reasoning.

Table 49 compares meta-planning with the model-based adaptation approach described in this dissertation. There are two key differences between meta-planning and model-based adaptation. The first key difference comes from the fact that meta-planning systems use operators and plans as their basic unit of representation and thus suffer from the same sorts of fundamental limitations that are imposed by these units as other planning mechanisms (as discussed in Sections 8.1.1, 8.1.2, and 8.1.3). The second key difference comes from the fact that meta-planning systems contain a fixed set of reasoning mechanisms and thus their knowledge about processing is built in to the planner and restricted to those mechanisms. Specifically, MOLGEN's mechanisms are heuristic search and least-commitment planning, while PAM's mechanisms include various forms of plan reuse and generation. The existence of a fixed set of strategies is both a benefit and a drawback to the meta-planning approach. The benefits of the fixed strategies lie in the fact that representations of strategies are broadly applicable; in contrast, anyone developing agents in REM or SIRRINE must provide a separate model of the reasoning mechanisms for every agent. The drawbacks of the fixed strategies lie in the fact that other forms or reasoning are not supported. Thus, for example, someone building an meta-reasoning agent which is guaranteed to only ever reason by heuristic search and least-commitment planning may wish to use the meta-planning mechanism in MOLGEN. However, someone building a meta-reasoning agent which involves a broader or simply different variety of reasoning techniques would be better served by REM or SIRRINE.

The reasoning architecture used in Guardian and a variety of related agents [Hayes-Roth, 1995] is another system which uses planning to guide planning. Unlike PAM and MOLGEN, however, planning operators in Guardian's

architecture are much more elaborate than traditional planning operators. Operators in that architecture include not only requirements and results but also tasks involving perception, prioritization, additional planning, etc. These operators enable adaptive reasoning because the effects of one operator can influence the selection and scheduling of other operators. The most substantial difference between REM and the architecture used in Guardian is that the adaptation operations in the latter are defined within the agent using the language of the architecture. In contrast, the adaptation processes in REM are defined within the architecture (they are also defined in the language of the architecture; REM does contain a TMKL model of its own reasoning, including its adaptation strategies). This difference has considerable implications for the kinds of adaptation performed. Adaptations in Guardian are very simple, involving results like switching a mode. Furthermore, they are very specialized to a particular kind of situation in a particular domain. However, there are a great number of these adaptations and they are invoked frequently and rapidly during execution. In contrast, adaptation in REM involves large, complex processes which can make many changes to diverse portions of the agent. REM has only a few adaptation strategies and some of these strategies can be very expensive. However, the adaptation strategies that REM does have work for a broad variety of domains and problems.

The architecture for Guardian seems particularly well suited to Guardian's domain: life support monitoring. This domain is complex and dynamic enough that adaptation is very important. It is also a domain that demands great speed and reliability; adaptation must be performed with minimal computation and the particular kind of adaptation performed must be one which has been tested or proven to be correct. It is not acceptable for Guardian to just try some sort of adaptation and see if it works out. Specialized, domain-specific adaptation operations seem to be the most effective way of obtaining this behavior. In contrast, the web browsing domain, for example, not only allows an agent to try out new kinds of behaviors but even demands that an agent do so. It is virtually impossible to build a set of pre-specified adaptations which handle the entire diversity of circumstances a web agent could encounter. Even if one did build such a set of adaptations, it would rapidly become obsolete as the web changes. Fortunately, web browsing does not require the speed and reliability that Guardian's domain does. Thus the slower and less certain but more generic and more dramatic adaptation mechanisms in REM are appropriate.

Another area of research on meta-reasoning involves allocation of limited processing resources. For example, MRS [Genesereth, 1983] is a logical reasoning architecture which uses meta-rules to guide the ordering of inference steps. As another example, anytime algorithms [Boddy and Dean, 1989] perform explicit deliberation-scheduling to allocate processing time to a set of algorithms which make incremental enhancements to an existing solution to a problem. Similarly, the Rational Meta-Reasoning approach [Russell and Wefald, 1991, Chapter 3] computes the expected benefits of applying algorithms versus simply acting. Table 50 presents a comparison between model-based adaptation and meta-reasoning for allocation of processing time. Model-based adaptation focuses on functional issues, i.e., changing what the agent does. In contrast, processing allocation techniques focus on non-functional issues, i.e., changing how and when the agent does what it does. Consequently, the two perspectives are more complementary than competing, and there is a potential for future work to explore the synergy between functional and non-functional adaptation. This idea is explored in more detail in Section 9.1.1.

8.1.7 Agent Modeling

The field of agent modeling provides an important perspective on the question of what information provides a useful representation of a process. The primary difference between the work in this dissertation and existing work on agent modeling is that this dissertation uses agent models for automated self-adaptation to perform previously unknown tasks. Most other agent modeling projects focus on non-automated (more precisely, semi-automated) processes. Furthermore, many of these processes do not focus on adaptation. For example, CommonKADS [Schreiber et al., 2000] and DESIRE [Brazier et al., 1997] are methodologies for building knowledge systems which use models of agents throughout the development process. In contrast, ZD [Liver and Allemang, 1985, Allemang, 1997] does support extensive automated reasoning, but this reasoning emphasizes analysis rather than adaptation.

Table 51 summarizes key differences between the models used in CommonKADS and the TMK modeling paradigm of which TMKL is an example. A representation of a knowledge system in CommonKADS is divided into several different models, such as an organization model, a task model, a knowledge model, and a design model. This division is similar to the division in TMK between tasks, methods, and knowledge. In fact, the models in CommonKADS seem to resemble TMK fairly closely. The most dramatic difference between CommonKADS models and TMK is the fact that the former rely heavily on relatively informal sorts of representations such as diagrams and natural-language text. In contrast, most forms of TMK models (including TMKL) are much more formal. This is directly motivated by the fact that TMK is designed to facilitate automated reasoning.

	Processing Time Allocation	Model-Based Adaptation
Why does meta-reasoning occur?	To schedule both reasoning and action	To alter the effects of an existing reasoning strategy
What meta-knowledge is being used?	Knowledge about the expected benefits of reasoning over time	Functional knowledge about reasoning mechanisms and the domain
How does meta-reasoning occur?	A variety of utility-based decision making techniques	Redesign of models
When does meta-reasoning occur?	Whenever a decision about what process to perform next occurs	Before or after an entire episode of reasoning
Summary	The main difference between model-based adaptation and processing time allocation is that the former focuses on meta-reasoning to alter the functionality of a computation while the latter focuses on controlling the use of time in a computation. This difference in purpose leads to very different representations and processes.	

Table 50: Comparison of meta-reasoning for allocation of limited processing time versus model-based adaptation.

	CommonKADS models	TMK models
Why are agents modeled?	To facilitate principled system design by humans	To provide knowledge for automated adaptation
What is encoded in a model?	Semi-formal representations of knowledge, functionality, and behavior, all in an organizational context	More formal representations of the knowledge, functionality, and behavior of a specific agent
How are models constructed?	By human experts	By human experts and/or automated reasoning processes
When are models constructed?	Before implementation with revisions throughout the life-cycle	Before implementation or (automatically) in the context of a particular reasoning session
Summary	CommonKADS models contain many of the same sorts of information as TMK models, but do so in a less formal way to support of human knowledge system engineering rather than automated adaptation.	

Table 51: Comparison between the models used in the CommonKADS methodology and TMK models.

There are also two major differences between the content of in CommonKADS models and the content in TMK: CommonKADS' emphasis on organizations and TMK's emphasis on teleology. The first of these involves the fact that CommonKADS situates it's information about a system in the context of information about the organization which uses the system. For example, if the meeting scheduling agent described in Section 4.4 were described in the CommonKADS notation, that description would include an organization model which would contain facts about the significance of meetings, what sort of people have the power to authorize meetings, what sorts of resources are required to hold a meeting, what sorts of restrictions the organization places on meetings, etc. The TMK model of this agent in SIRRINE does have some information of this sort; for example, it does represent the meeting scheduler's knowledge that meetings can ordinarily only be held during business hours. However, there is a difference between modeling an agent's place in its organizational structure (as in CommonKADS) and modeling only the knowledge that the agent has about its place in that structure (as in TMK). An agent may know very little about its role in an organization, but the designer of an agent may need to know much more about that role. The reflective reasoning paradigm addressed in this dissertation demands that the knowledge needed to redesign an agent be encoded in that agent (so that the agent can redesign itself). Thus the kinds of information which appears in organization and related models in CommonKADS may be useful to encode in the "K" portion of a TMK model.

The other major difference between the content of CommonKADS models and the content in TMK is the focus on teleology in TMK. TMK models are both functional and teleological; they represent the overall purpose of an agent and they represent how the purposes of the components of an agent combine to accomplish that overall purpose. CommonKADS models are clearly functional; they provide extensive information about the purposes to which an agent is applied. However, models in CommonKADS provide much less insight into the teleology of an agent than TMKL models do. For example, a CommonKADS task model of the meeting scheduling agent would include a description of the organizational tasks which the agent serves (in this case, the one task that it serves is scheduling meetings). The TMK model of the agent also represents this task. However, unlike CommonKADS, TMK also represents tasks which are internal to the agent, e.g., checking a particular slot against a particular schedule. The representation of the internal portions of a system in CommonKADS appears in the design model. The CommonKADS design model is purely structural and thus does not provide a detailed view of the internal tasks being addressed. This structural view provides relatively little information about a system's teleology. This is not a critical limitation for the purposes to which CommonKADS is applied because a human reader can infer the teleology of a system from its overall functions and its structure. However, for an automated reasoning system it is very helpful to have that information explicitly encoded. For example, the relation mapping mechanism within Section 6.3 and the diagnosis mechanism within Section 6.4.1 both critically depend on the knowledge of the functions of components of the agent.

ZD is another framework for modeling agents. Table 52 provides a comparison between TMKL and ZD. These languages are very similar; in fact, much of the early work on TMKL [Murdock, 1998a] focused on integrating the TMK formalisms from SIRRINE and earlier systems with ZD. TMK and ZD comprise very similar perspectives on the modeling of agents. They were motived by a strongly overlapping (but not identical) set of issues and previous work.[1] However, the use of ZD was focused primarily on analysis and validation, rather than on execution and repair, which motivated many of the major differences between the approaches. TMKL ultimately resembles earlier TMK formalisms much more than it does ZD, largely because the applications which TMKL are intended for are more similar to the uses of TMK in SIRRINE, etc. then the uses of ZD. However, there are several features of TMKL which were inspired by ZD. For example, the explicit identification of incidental consequences of a process (i.e., the additional-results slot in a method, as described in Chapter 2) is derived from a similar mechanism in ZD. Also, the combination of states and transitions in a TMKL method much more closely resemble the analogous portions of ZD than they do the analogous portions of earlier TMK models. The capabilities that TMKL provides and ZD does not mostly relate to the fact that TMKL models are integrated with working code. This is necessary for TMKL because they are run by REM's execution process (from Chapter 5) and the adaptation of the model (described in Chapter 6) needs to directly result in a change in the actual behavior of the agent.

8.1.8 Credit-Assignment

A key aspect of model-based adaptation is determining what portions of a system are responsible for some characteristic of that system. In the work presented in this dissertation, there are three major forms of credit-assignment. One of these forms occurs during proactive model transfer; the first portion of the relation mapping algorithm in Section

[1] More on the history of TMK appears in Section 8.2.1.

	ZD	TMKL
Why are agents modeled?	To provide knowledge for automated analysis	To provide knowledge for automated adaptation
What is encoded in a model?	Knowledge, functionality, and behavior of a proposed or working software system	Knowledge, functionality, and behavior of a working software system, including links to the code
How are models constructed?	By human experts	By human experts and/or automated reasoning processes
When are models constructed?	Before implementation with revisions throughout the life-cycle	Before implementation or (automatically) in the context of a particular reasoning session
Summary	ZD and TMKL are very similar in nature. The most significant differences between the two are a consequence of the fact that ZD is primarily used to analyze a design (either of a working system or a proposed one) while TMKL is used to modify a working system.	

Table 52: Comparison between the ZD and TMKL modeling languages.

6.3 identifies of a set of candidate tasks which could be modified from the model for the existing task to contribute to the new task. Another form of credit-assignment occurs during failure-driven model transfer; the feedback analysis algorithm in Section 6.4 searches through a trace to find steps in which the actual behavior contradicts the desired behavior specified in user feedback. The third form of credit-assignment in this work occurs during execution; checks are made for various kinds of failures for which the model enables identification (e.g., when a task specifies a makes condition and that condition does not hold when the task is completed).

All three of the types of credit-assignment in model-based adaptation involve explicitly testing whether some desired result matches some existing result. Thus they can all be classified as forms of discrepancy detection [Davis, 1984]. Note, however, that there are some significant differences between credit-assignment in the work presented in [Davis, 1984] and the work presented here. The most obvious are the fact that the former involves detection of failures in the domain of electric circuits while the latter involves detection of adaptation opportunities in an intelligent agent. A more subtle difference is the fact that the models which are analyzed in the former work involves only structural and behavioral information while the latter involves structural, behavioral, and functional information. The inclusion of functional information within a model of a system is a controversial issue in this research area [de Kleer and Brown, 1984, Keuneke and Allemang, 1989]. The inclusion of functional information within a model of a reasoning system seems particularly valuable, because the structure and behavior of such systems can be overwhelmingly complex. Each line of source code in a program has its own structural and behavioral characteristics; inclusion of function within a model provides abstractions which identify only *relevant* aspects of behavior and thus allow credit-assignment to focus on those aspects.

An alternative to functional models for credit-assignment of intelligent agents involves restricting credit-assignment to only a restricted portion of the agent (by assuming that all other portions of the agent cannot be changed). For example, TEIRESIAS [Davis, 1979] performs user-supported credit-assignment on a set of rules in an expert system to determine bugs; its representation of these rules contains only behavioral information (what they do) and no functional information (why they are there). This approach is feasible, at least to a limited extent, because it assumes that the basic algorithm (backward-chaining) is fixed and thus restricts its credit-assignment to the rules. It would be extremely impractical to consider all possible portions of both the rules and the computations which use those rules. In contrast, REM is able to reason about the process and content of an entire agent in an integrated manner because functional information (encoded in the task portions of the model) is used to guide the assignment of credit.

While TEIRESIAS focuses on facilitating user editing of a rule base, it is also possible to automate credit-assignment and modification of rules. CASTLE [Freed et al., 1992] performs detection focusing and threat detection to monitor for failures during execution and then uses a justification structure to search for a localization which can support some sort of change. Note that the justification structure is a record of inferences which lead to a result much like the meta-knowledge in a TMS (as discussed in Section 8.1.5). However, unlike a TMS, CASTLE uses the justification not merely for maintenance of knowledge but also for the development of new rules to manipulate that knowledge. The most dramatic difference between CASTLE and the failure-driven model transfer mechanism

presented in Section 6.4 is that the former is able to operate with very little abstract knowledge because it restricts credit-assignment to rules alone while the latter requires more complex representations because it considers the all portions of a reasoning process as a potential source of credit.

8.2 Reasoning with TMK Models

A key aspect of the research reported in this dissertation is the representation of processing in terms of tasks, methods, and knowledge. This basic idea is not new to this dissertation. Section 8.2.1, below, provides a brief overview of the history of this approach. Section 8.2.2 then describes the key contributions that the work presented in this dissertation makes to this line of research.

8.2.1 History of Reasoning with TMK

Two key ancestors of TMK are Generic Tasks [Chandrasekaran, 1988] and Functional Representation [Sembugamoorthy and Chandrasekaran, 1986]. The basic idea behind Generic Tasks is that reasoning processes can be classified and combined by identifying the kinds of tasks being addressed. Encoding reasoning in terms of tasks involves representing potentially complex subsystems. Thus this approach contrasts strongly with approaches which only represent small atomic pieces of computation (e.g., production rules, planning operators, etc.). The basic idea behind Functional Representation is that devices can be analyzed as a combination of functions, which illustrate intended effects of devices and behaviors, which involve the sequences of events which result in effects.

One piece of work which came out of the this line of research was Kritik [Goel, 1989], a physical device design system which used Structure-Behavior-Function (SBF) models of devices. SBF models build on the relationship between functions and behaviors developed in Functional Representation, but are also influenced by other work on representation of physical devices (particular with respect to information about primitive components, substances, etc.).

Kritik lead to a variety of key developments in this line of research. The representation of SBF models was substantially elaborated and formalized in Kritik2 [Goel et al., 1997b]. This more elaborate form of SBF also enabled much more sophisticated reasoning in IDeAL [Bhatta, 1995], including automated abstraction of generic mechanisms. While this work on SBF was occuring, research on Router [Goel et al., 1994], a navigational planning system, explored multi-strategy reasoning in which tasks could be addressed by a variety of methods and explicit criteria were used to select among methods. This approach was derived from the Generic Tasks paradigm.

The two branches of this research, Kritik and Router, came together in the Autognostic project [Stroulia, 1994, Stroulia and Goel, 1995, Stroulia and Goel, 1997]. The basic idea behind Autognostic is that task-based problem solving systems such as Router can be viewed as computational devices and thus can be redesigned using the kinds of representations and reasoning used in Kritik2. Autognostic has a language for modeling problem solving in which tasks are viewed as analogous to functions and methods are viewed as analogous to behaviors. These models were originally called SBF models of problem solvers (e.g., in [Stroulia, 1994]) but later papers refer to them as SBF-TMK models (e.g., in [Stroulia and Sorenson, 1998]). Autognostic takes as input a problem solver, encoded in this modeling language, plus a problem to be solved. It runs that problem solver and if it is unable to solve the problem or if (after the process is done) the user provides feedback recommending an alternative solution, Autognostic modifies the solver.

Most of the earliest uses of the term "TMK" to describe models of tasks, methods, and knowledge appeared in descriptions of Interactive Kritik [Goel et al., 1996, Goel and Murdock, 1996]. Interactive Kritik was developed as an extension of Kritik2; it adds a graphical user interface to explain the process and products of the design reasoning in Kritik. The graphical explanations of process in Interactive Kritik are organized around tasks, methods, and knowledge; the approach is similar to the models of problem solvers in Autognostic but is much less formal and elaborate since it is only used to present reasoning, not to direct or modify reasoning. Interactive Kritik was used in the HIPED [Murdock et al., 1998] project, which explored the integration of task-method structures with database systems (thus providing another perspective on reasoning at the level of tasks and methods).

There is also a body of relatively recent work on TMK which has focused on integration with software engineering technology; the idea behind this integration is that even if a system was not built to be a multi-strategy problem solver, it can at still be analyzed in terms of what it does and how it does it (and thus tasks and methods can be ascribed to it). One instance of this sort of work involves the integration of SBF-TMK models with software architecture descriptions [Stroulia and Sorenson, 1998]. Another instance of this trend is MORALE [Abowd et al.,

1997], a large, multi-faceted project involving development and interoperation of a wide variety of tools for evolving legacy software systems based on software architectures. Some of the work described in this dissertation (particularly work on SIRRINE) was performed as part of the MORALE project.

In addition, there has been a recent interest in using TMK models for representing human reasoning processes. ToRQUE [Griffith and Murdock, 1998] used a version of TMK based primarily on Autognostic's SBF-TMK models to guide both selection and abandonment of methods in the context of modeling human scientific reasoning; the tasks, methods, and knowledge in ToRQUE are based on analyses of transcripts of actual human problem solving episodes. DESPINA [Davies, 1999] also involves modeling human reasoning in TMK; DESPINA uses SIRRINE for representing and executing its reasoning model.

There are several ongoing projects relating to TMK. One of these projects is the Ectropic Software project [Rugaber and Guzdial, 1999], which, in part, builds on some of the work which is reported on in this dissertation. This project is considering the use of design representations for ensuring that software does not become overwhelmingly disorganized during the course of development and evolution. The representations being developed in this project are influenced by a variety of past work, including the integration of TMK and software engineering technology in the context of the MORALE project.

8.2.2 Contributions to Reasoning with TMK

Many of the projects described in the previous section only use fixed TMK models; i.e., no changes to the models are ever made. For example, Interactive Kritik has an informal graphical TMK model built into its interface, but it has no formal internal representation of that model so it cannot do any reasoning about the model. ToRQUE does have an internal representation of the TMK model, but it does not make any changes to the model. It does, however, use the model not only for selection of methods (as in Router) but also for abandonment of methods which do not appear to be making progress. These systems make a significant contribution to the general theory of reasoning with TMK, but this contribution is very different from the work in this dissertation which focuses on the adaptation of TMK models.

Autognostic has substantially more in common with this work than those other TMK projects do. Table 53 provides a summary of the major differences between Autognostic and REM; SIRRINE lies somewhere between the two, having some of the characteristics of Autognostic and some of the characteristics of REM. The first difference listed, why adaptation occurs, is more a matter of perspective than an absolute division. Some sorts of adaptation can be alternatively seen as repairing a failure (as in Autognostic) or adding new capability (as in REM). The key distinguishing characteristic is whether a capability *should* have been included in the original agent; Autognostic fixes agents which do not do what they should have while REM has agents do things which are outside of what they are currently supposed to do. This distinction can be subjective, and some adaptation problems can be viewed either way. However, this difference in perspective is still a significant one because it affects the way that these systems address these challenges. For example, the fact that Autognostic only performs modifications after execution is a direct reflection of its specific purpose; a fault in a system cannot be repaired until after there is evidence that there is a fault. In contrast, because REM allows a user to request that a new (unimplemented) task be addressed, it can be required to perform adaptation before it can even attempt execution.

There are several other aspect of REM which are significantly different from Autognostic. Adaptation in Autognostic involves making a single change to an agent (although this change may be complex). The single change assumption is often reasonable for failure-driven learning since many failures have a specific cause. However, it is much less reasonable for adapting for new tasks; there is no particular reason to believe that a new task can be addressed by making exactly one change to the process for an existing task (even if the two tasks are similar). REM allows multiple changes to diverse, distant portions of an agent architecture.

REM also provides an integration of pure model-based techniques with both generative planning and reinforcement learning. This integration provides substantially greater problem-solving coverage than model-based techniques alone (while avoiding much of the computational cost of these other techniques used by themselves). This expanded coverage is particularly valuable in providing new capabilities because existing tasks and methods may be completely irrelevant to the new task *or* to some piece of a revised process for an existing task. This possibility also exists for addressing failures, but it is reasonable to expect that it is much less likely; one would expect that if a capability should have been in the agent to begin with that the existing functionality is at least close to the desired functionality. However, this expectation is not reasonable when adding new capabilities. Certainly, if new functionality is *never* close to the existing functionality then there is no point to having model-based adaptation at all; one should simply use a technique, such as generative planning or reinforcement learning, which reasons without any prior strategic

	Autognostic	REM
Why are agents adapted?	To repair agents which fail to address the task for which they were designed	To add new capabilities to an agent.
What portions of adaptation are done by the agent?	Models are automatically modified, but any changes to primitive tasks must be done by a programmer.	Modifications are made to both the models *and* the internal mechanisms of the primitives.
How are models adapted?	Through analysis of a trace and a model	Through analysis of a trace and a model, and through analysis of a model alone, and through generative planning, and through reinforcement learning
When are models adapted?	After execution	Before and after execution
Summary	Autognostic and REM both make changes to TMK models. However, the adaptation in REM addresses a different set of issues and approaches to address a subtly different problem: adding incrementally new capabilities rather than repairing existing capabilities.	

Table 53: Comparison between the Autognostic and REM modeling languages.

knowledge. REM is particularly well-suited to environments in which the mechanisms for new requirements are *usually* similar to existing mechanisms (allowing model-based adaptation) but occasionally there are no relevant mechanisms for a task or subtask (demanding reinforcement learning or generative planning).

One key aspect of the differences between Autognostic and REM lie in their formalizations of TMK. REM's TMKL provides much more expressive representation of concepts, relations, and assertions than the TMK language in Autognostic; this additional power is derived from the fact that TMKL is built on top of Loom. The added capabilities of Loom are very useful in adaptation for new tasks, since reasoning without traces often demands more elaborate knowledge about domain relationships and their interactions with tasks and methods. The relation mapping algorithm in Section 6.3 provides a particularly substantial example of how knowledge of the domain can be used in adaptation.

TMKL also provides a more elaborate account of how primitive tasks are represented than the TMK formalism in Autognostic does. Primitive tasks in Autognostic always include a link to a LISP procedure which implements the task; because Autognostic cannot inspect or reason about these procedures, it cannot make changes to them. Thus any modifications to the building blocks of the agent in Autognostic must be done by a programmer. TMKL does allow primitive tasks to include a link to a LISP procedure, and, like Autognostic it is unable to modify these procedures. However, TMKL also allows other sorts of primitive tasks, which are defined using logical assertions or parameter value bindings (as described in Section 2.2). These representations can be directly manipulated and modified by REM's adaptation processes. Some sorts of primitive functionality may not be adequately represented by these sorts of primitives in which case they must be represented in LISP and cannot be adapted by the system. However, the existence of some directly manipulable primitives means that REM can alter some of the primitives in an agent and can also add new primitives to interact with the ones which it cannot manipulate. Consequently, it is possible for agents in REM to change both models and primitive tasks within the models, thus performing completely automated self-adaptation.

8.3 Research in Cognitive Science

The research described in this dissertation focuses on the development of theories of intelligence; the question of whether these theories are of how humans *do* reason or of how computer programs *can* or *should* reason is viewed as secondary (though still important). The particular research described here uses development and analysis of software systems as its primary experimental method. Consequently, the results obtained here are more directly informative about artificial reasoning than about human cognition. However, there do seem to be some potential insights to the topic of Cognitive Science in this work. Furthermore, there are also insights which come from Cognitive Science and have been useful in building this theory. For example, consider the distinctions between internal versus external and

state versus definition relations described in Section 2.4.2. These distinctions categorizes relations in terms of how they can be accessed and manipulated by the agent. This categorization is inspired, in part, by work on ecological human cognition which emphasizes issues of access and manipulation [Kirlik, 1998]. Knowledge of this sort has proven useful for a variety of purposes in REM, e.g., in identifying what sorts of information can be directly mapped to related information, as in Section 6.3.

The key motivation for the claim that this work makes contributions back to the study of human cognition is the idea that a truly general and powerful computational theory of intelligence is inherently a useful and interesting hypothesis regarding human cognition.[2] Consider the space of all *possible* mechanisms for representing and manipulating self-knowledge. It is clear that this space is extremely large; virtually any kind of knowledge and inference could potentially be in memory. In contrast, consider the space of all such mechanisms which are adequate to perform intelligent reasoning. This space is clearly far more restricted; the mechanism must be shown to be both powerful enough to allow intelligent reasoning to occur, parsimonious enough to form a credible theory, and general enough to support a wide range of intelligent behavior. To the extent that these are strong restrictions, the space of adequate solutions is very small with respect to the space of possible solutions. Furthermore, as the breadth and depth of the demands are increased, it can be expected that the space of adequate solutions becomes so small that any adequate solution approximates any other adequate solution. There is little reason to believe that an arbitrary AI process which simply addresses a single task in a single domain does so in a way that resembles the way that humans do. However, as the range of capabilities of an approach increases, it becomes increasingly likely that the combined demands of the problems force a solution which bears an increasing resemblance to the way humans perform similar reasoning. Thus to the extent that this theory has been shown to provide a broad and interesting range of capabilities, it constitutes a reasonable hypothesis for a cognitive model.

A deep investigation of these claims of relevance to human cognition is beyond the scope of the research described here. There is some existing and ongoing work which does provide evidence of the cognitive plausibility of TMK models and the model-based reasoning techniques which use them, e.g., [Griffith et al., 1997, Griffith and Murdock, 1998, Murdock, 1998b, Davies, 1999]. To the extent that the representations and reasoning in REM and SIRRINE are extensions of this broader theory, this provides us with further evidence that the research here has some cognitive validity.

8.4 Research in Software Engineering

Another major area of research which the work in this dissertation relates to is Software Engineering. The primary goal of Software Engineering is the creation and modification of software. The intelligent reasoning processes that are modeled, executed, and adapted in SIRRINE and REM are a form of software. Thus one can view the adaptation processes in this research as an automated form of Software Engineering. Consequently, it is important to consider how existing techniques within the field of Software Engineering relate to this work.

One Software Engineering topic in which is particularly relevant to this work is Software Architectures. There are a variety of Architectural Description Languages (ADL's) which provide representations of software systems. TMKL also provides a representation of software systems. Table 54 provides a list of key differences between the ADL's and TMKL. In general, the primary difference between software ADL's and functional modeling languages is that the former focus on structural descriptions of software (primarily components and connections) while the latter provide both structural and functional descriptions [Clements, 1996]. TMKL is similar to other functional modeling languages (such as ZD, described in Section 8.1.7) in this regard.

Some of the work in this dissertation has further explored the relationship between Task-Method-Knowledge modeling and Software Architectures. In particular, there is a mechanism within SIRRINE which exports TMK models into ACME [Garlan et al., 1997]. ACME is a general framework for representing architectural information; unlike traditional ADL's it does not make a strong commitment to a particular ontology. Instead it provides a few basic features (such as components and connectors) and a mechanism for allowing additional kinds of information to be defined. ACME is primarily intended to act as an interchange format for information encoded in various ADL's. The translation in SIRRINE from TMK to ACME provides the potential for information from SIRRINE to be used in a wide variety of Software Engineering tools. This potential has been tested by importing SIRRINE generated

[2]Note that this claim is fundamental to the role of AI and computational modeling in the study of human cognition. The argument presented here is present, either implicitly or explicitly, in an enormous range of research, e.g., [Newell and Simon, 1963, Minsky, 1975, Hayes-Roth and Hayes-Roth, 1979, Anderson, 1983]. Furthermore, it is largely equivalent to traditional philosophical views on the nature of all scientific theories, e.g., [Kant, 1781].

	Software Architectures	TMKL
Why is software represented?	To facilitate principled software design by humans	To provide knowledge for automated adaptation
What information is represented?	System structure	System structure and function
How are models constructed?	By human experts	By human experts and/or automated reasoning processes
When is software represented?	Before implementation with revisions throughout the life-cycle	Before implementation or (automatically) in the context of a particular reasoning session
Summary	Architecture Description Languages (ADL's) typically involve a comparable level of abstraction to TMKL models but generally focus on structural information (components and connections) and not functional information. Some ADL's are less formal than TMKL models because they are generally intended to support human software engineers, not automated adaptation.	

Table 54: Comparison between Architecture Description Languages from Software Engineering and TMKL models.

	Java Reflection	Model-Based Reflection
Why does reflection occur?	To allow a system to access its classes	To alter the functionality of an intelligent agent
What is reflected on?	Information about classes	Information about both knowledge *and* process
How does reflection occur?	The language enables access to the information	The shell provides a variety of automated adaptation strategies
When does reflection occur?	During execution	Before and after execution
Summary	Java provides some features for performing reflection, but those features simply provide the ability to access data structures (classes). In contrast, REM and SIRRINE reason about both the data and the process, and provide entire adaptation strategies for altering reasoning in response to new requirements.	

Table 55: Comparison between the reflection capabilities in the Java programming language and the model-based reflection capabilities described in this dissertation.

ACME descriptions to VisEd [Waters, 1999b], a visual editor for architectural descriptions; this conversion is enabled by ACMEServer [Waters, 1999a], an ACME based information exchange tool.

Another topic in software engineering which relates to this dissertation work is reflection. The Java programming language provides a collection of features for performing reflection [McCluskey, 1998]. Table 55 provides a comparison between reflection in Java and the reflection in REM and SIRRINE. The reflection features in Java provide the ability to access information about the classes of the agent. Classes in Java resemble concepts in Loom (and thus in TMKL), so these reflection features serve a similar purpose to the various concept accessing features of Loom. Note, however, that a Java class can also contain methods, i.e., pieces of code which accomplish effects using the data in the class. Thus to the extent that Java reflection involves these classes, there is also a limited amount of procedural information which can be accessed via reflection. However, unlike TMK models, this information does not include function *or* composition; consequently, it does not appear to be well-suited to directly enabling the sorts of model-based reflection performed by REM and SIRRINE. If one were to build a new variation on REM and SIRRINE that worked within Java, it seems likely that the Java reflection capabilities would be useful, but they would certainly not provide all of the necessary representation; much of the information currently found in TMKL models would have to be explicitly encoded on top of the reflection capabilities provided.

There are more elaborate notions of reflection in the Software Engineering field. For example, Reflective Object Oriented Analysis [Cazzola et al., 1999] involves the use of meta-objects to denote certain kinds of non-functional properties. Table 56 presents a comparison between ROOA and TMKL-based reflection. The basic idea behind

	Reflective OO Analysis	Model-Based Reflection
Why does reflection occur?	To provide encapsulation of non-functional traits	To alter the functionality of an intelligent agent
What is reflected on?	Meta-classes involving non-functional traits	Models of knowledge and functional and behavioral aspects of processing
How does reflection occur?	Meta-classes can be instantiated and used just like base-level classes	Adaptation is supported by automated reasoning strategies within the shell
When does reflection occur?	During execution, as the non-functional traits become relevant	Before and after execution
Summary	Reflective Object Oriented Analysis provides a framework for explicitly representing non-functional traits. The model-based reflection described in this dissertation focuses on representing and reasoning about functional aspects of an agent. Thus the two approaches seem largely complementary.	

Table 56: Comparison between Reflective Object Oriented Analysis and model-based reflection on TMKL models.

ROOA is that certain kinds of data and actions are specifically suited to enabling non-functional traits across a wide variety of functional elements. For example, persistence is a trait which demands capabilities such as storage and retrieval and involves data such as size and location in persistent memory; thus ROOA recommends that these capabilities and data be encapsulated in a relatively abstract meta-object which is independent of the functional requirements of the lower level objects which depend on it. Other sorts of non-functional requirements which are supported by ROOA include fault-tolerance and security. In contrast, the model-based reflection techniques presented in this work deal entirely with functional adaptation; all of the reflective reasoning is done in the service of performing new tasks or new variations on existing tasks. The addition of non-functional capabilities to model-based reflection seems like an interesting topic for future research; see Section 9.1.1 for more on this subject.

Chapter 9

Future Work

The topics of reflection, models, and adaptation are very broad and there is a virtually limitless range of future work which can be done in this area. This chapter focuses on ideas for future work which have a particularly direct connection to the research described in this dissertation. The first section focuses on enhancements to the existing capabilities of REM and SIRRINE while the second section describes some additional capabilities which could potentially be added to these reasoning shells. Finally, the third section abstracts and summarizes the key points that this discussion illustrates about the applicability of the existing research.

9.1 Enhancements to Existing Capabilities

Future work could extend the existing capabilities of REM and SIRRINE in a variety of ways. The four subsections of this section describe potential enhancements to the work described in Chapters 2, 3, 5, and 6 respectively.

9.1.1 Models

The research described in this dissertation has established that TMKL can be used to encode a wide range of agents and support a wide variety of adaptations. However, there certainly are some sorts of capabilities that TMKL does not provide. One major limitation of TMKL is its lack of any representation of non-functional characteristics such as efficiency, reliability, security, etc. This limitation hasn't proven to be a major difficulty in enabling the kinds of adaptations of the kinds of agents that have been studied in this research. Note, however, that the adaptation problems which these systems have been given have all been functional in nature; it seems likely that TMKL in its current form would be poorly suited to performing non-functional adaptation. For example, while REM is able to transform an disassembly agent into an assembly agent, it seems unlikely that it would be able to transform a disassembly agent into a faster or more reliable disassembly agent.

It is possible to translate some sorts of non-functional requirements into functional requirements; this would allow these requirements to be encoded in REM. For example, if someone had a disassembly agent which was guaranteed to take no more than ten minutes to disassemble a device, one could represent this fact by having the makes state of the main task require that the device be disassembled and that the elapsed time be less than ten minutes. One

could then have various subtasks also include the effect that they have on the elapsed time in their description. If this were done then REM, in its current form, would be able to directly address some sorts of adaptations involving speed. However, there are several reasons why this is not an entirely satisfactory answer to this problem. One of these is that this approach does not take into account the time taken by the execution and adaptation processes in REM; for example, if a user wanted to adapt the disassembly agent which takes 10 minutes into one which takes 8 minutes, REM might be able to produce a version for which the total cost of the primitive actions is 8 minutes but it might take an extra 5 minutes doing the adaptation and monitoring the execution.

Furthermore, representing non-functional traits such as elapsed time as ordinary logical assertions forces a system to only reason about those traits using mechanisms that are appropriate to any assertion. These mechanisms are unlikely to have the power that specialized techniques would have. If information about elapsed time were made an explicit part of the modeling language, it would be possible to add adaptation mechanisms to the reasoning shell which are particularly suited to reasoning about and manipulating elapsed time. There are a wide variety of existing mechanisms for reasoning about reasoning time, e.g., some of the meta-reasoning approaches discussed in Section 8.1.6. A new modeling language which provided capabilities for representing time could potentially allow these or similar techniques to be combined with the functional model-based reflection approach described in this dissertation to provide a more general theory of adaptation.

9.1.2 Traces and Feedback

The representation of traces in REM and SIRRINE are a fairly direct reflection of the representation of models in those systems. Thus potential enhancements to the models in those shells could lead to comparable enhancements to the traces; for example, if a model contained references to how long a task should take to execute (as described in the previous section) then it may be useful for a corresponding trace to contain a record of how long that task actually did take to execute.

Another topic for future research regarding traces is the idea of producing a more compact trace. Currently, a trace records all invocations of tasks, methods, transitions, reasoning states as well as all the knowledge states which acted as an input or output for a task. This can take a lot of memory to store and a lot of processing time to search; if the agent contains loops or recursion, the number of steps in the trace may be much larger than the model itself. One technique which could potentially help control this size issue would be to represent loops by specifying the steps involved and the number of times that the loop was invoked. It is not clear, however, where one would put all of the knowledge states involved in the loop in this case. The knowledge states are a vital portion of the trace when feedback is given because they allow the credit-assignment process to distinguish between the actual knowledge items produced and the items which should have been produced. Thus omitting all of the knowledge states within a loop would negatively impact the ability to perform adaptation. On the other hand, including all of the knowledge states in a loop ensures that the size of the trace within a loop be proportional to the number of iterations, so any savings by having a more compact representation of tasks and methods would reduce the total size of the trace by at most a constant factor. It may be possible in future work to form some sort of compromise in which some but not all of the knowledge states within a loop are recorded.

Feedback is an issue which seems to present many opportunities for future research. SIRRINE has a particularly limited mechanism for handling user feedback. A feedback item in SIRRINE must always be a knowledge state, i.e., a set of mappings between parameters and examples. For example, the feedback to SIRRINE in the web browser / PDF example (described in Section 6.4.2) is a set containing one mapping from the command parameter to the "acroread ~a" command. The meaning of this feedback is that some part of the most recently completed execution should have produced that value for that parameter.

The formalism that REM uses to represent feedback is much more powerful than the one used by SIRRINE. However, REM has only slightly more powerful mechanisms for *using* feedback than SIRRINE does; i.e., REM only uses a fraction of the power that its representation for feedback could potentially support. Feedback in REM comes in the form of an annotation on a trace; that annotation can contain an arbitrary logical assertion in the Loom framework. The meaning of the feedback is not only that the logical assertion holds but also that the user has stated that this logical assertion is particularly relevant to addressing a problem with the trace that it is attached to.

The primary use of feedback in REM is in the fixed value production repair mechanism for failure-driven transfer from Section 6.4.2. In this mechanism, the only type of feedback which is used is a single logical term involving the should-transform relation, described in Section 3.3.1, which maps a task to an input knowledge state and an output knowledge state; its meaning is that the given task should produce a result consistent with the output state if it is given an input consistent with the input state. There are three ways in which this form of feedback makes

a processing difference versus the feedback in SIRRINE. First, the fact that the feedback is attached to a trace means that it can be used to adapt the model in response to that trace even if other traces for that or other models have been generated more recently; in contrast, feedback in SIRRINE only ever refers to the most recent execution. Second, the fact the should-transform relation refers to an input state means that the user can restrict the adaptation to only operate under specific knowledge conditions. Third, the fact that the should-transform relation refers to a task means that the user has the option of *either* specifying that the result should have been produced by the task as a whole (which is assumed in SIRRINE) *or* that the result should have been produced by a specific subtask (which isolates credit-assignment for that feedback to that subtask and it's subtasks). The subtask form allows the user to provide assistance to the credit-assignment mechanism if the user already knows what portion of the process needs to be changed; if the user doesn't have that knowledge then the other form is also available.

Although the use of feedback in REM is broader that the use of feedback in SIRRINE, there are still many varieties and applications of user feedback which are available for future work. There are many potential variations on the should-transform relation which have not been considered in this work. For example, one could envision feedback which states that a task should produce one knowledge state *or* another. Furthermore, there are many possible sorts of feedback which don't directly relate to how a task transforms knowledge states. For example, one could envision feedback about the process (e.g., that a particular method should have been used) or about the quality of results (e.g., to provide a particular reward value for reinforcement learning). One could even envision feedback about non-functional characteristics (e.g., that the result should have been produced more quickly); such feedback could be used to support non-functional adaptation of the sort discussed in Section 9.1.1.

9.1.3 Execution

Much of execution is fairly straightforward and thus does not demand a great deal of additional research. The bulk of the execution algorithms described in Chapter 5 involve simply looking at what the model says to do and doing it. However, there are a few aspects of the model execution process for which more research could be done.

One potential source of future work in this area is the subject of traces. Section 9.1.2 describes some issues involving what to include in a trace. Another important issue is *when* to build a trace. The failure-driven adaptation strategies described in Section 6.4 demonstrate that traces are valuable sources of information for adaptation; both SIRRINE and REM always generate a trace whenever they execute an agent. However, the production of traces does also take time and memory; the value of the traces may not always justify this cost. For example, in domains in which failures are both rare and easily reproduced, it may be more efficient to not generate a trace during an ordinary execution but rather to wait until a failure is detected and then start the current process over, generating a trace for that execution. The idea of restricting explicit monitoring of reasoning to exceptional circumstances only seems to be consistent with claims from Cognitive Science about how and when humans do or don't consciously reason about their own reasoning [Neumann, 1984, Suchman, 1987]. Because humans seem particularly good at issues of flexibility and adaptation, these traits may be useful for enabling automated systems which address similar issues. The question when to generate a trace in a TMK-based reasoner is an important topic for future research.

One aspect of the execution process which could use a great deal of additional work is the decision-making mechanism described in Section 5.2. Recall that the primary idea behind that mechanism is that most decisions are guided exclusively by logical conditions within the model and that reinforcement learning is only used when those conditions do not uniquely determine an option. This idea has motivated the fact that the reinforcement learning mechanism in REM is very simple; the emphasis has not been on building an excellent reinforcement learning component but rather on keeping the number of decisions which that component needs to make so small that it doesn't need to be very good. This approach has been effective in the work that has been done so far, but improvements may be useful for future work.

One reason why the REM reinforcement learning mechanism is so limited is the fact that past work on adaptation of TMK models has focused exclusively on the support of purely model-based techniques and not reinforcement learning; TMKL was largely motivated by that past research and thus is much more heavily tuned to reasoning of that sort. Consequently, the reinforcement learning process over these models has essentially been forced to fit the representation rather than having a representation which was built to support it. Thus one way to improve that process would be to provide additional information in the models to explicitly support reinforcement learning. One kind of information which would be particularly valuable for reinforcement learning would be information about intermediate costs or rewards which occur during execution. Without intermediate rewards, it is not possible to do useful learning over early states in a process until learning has been done for later states. This adds considerably to the total cost of learning. The reinforcement learning mechanism in REM provides only a single reward at the end

if the desired result of the specified task has been accomplished (because this is the only information that TMKL provides about the desired overall effect of a process). Information about execution time, as discussed in Section 9.1.1, could provide some additional information to address this issue (by penalizing decisions that lead to long running time). Additional information about intermediate rewards could also be added to the TMKL formalism.

Another kind of information which would be very valuable for reinforcement learning would be information about how to recognize a state. Recall (from Section 5.2) that REM treats the current position in the model *plus* the entire history of decisions made by reinforcement learning during the current execution as a state. This approach has a serious drawback in that it creates a state space which is exponential with respect to the number of decisions made. This is a particularly severe problem because reinforcement learning can be very expensive with respect to the size of the state space. Unfortunately, the problems on which the REM reinforcement learning mechanism have been tested do not seem to work using simpler information from the TMKL model (e.g., the current position in the model). One solution would be to include a slot in those portions of the model in which a decision would be made such that the value in the slot is an expression which defines a state. This would allow the designer of the model to specify the state space, avoiding the automatic exponential explosion that occurs in the current mechanism.

The two potential additions to the modeling language described above (intermediate rewards and specialized state spaces) do add a significant extra knowledge requirement. This extra requirement is mitigated somewhat by the fact that the information would be optional; if the information were omitted, the reasoning shell could simply default to the behavior in REM. It is important to note, however, that the most significant uses of reinforcement learning in the experiments which have been conducted on REM have focused on using reinforcement learning on the results of an adaptation process (specifically, the resolution of possible changes made by the relation mapping algorithm). A more effective reinforcement learning mechanism which relied on extra information from the system's designer would not be useful on decisions which were imposed by an automated adaptation technique unless that adaptation technique could also provide that additional information. The question of what sorts of extra information a model-based adaptation technique could provide to facilitate reinforcement learning over its results is yet another potential topic for future research.

In addition to providing more information to support the decision making process, it may also be productive to consider different algorithms for doing the decision making. While Q-learning is a powerful and easy to use technique for decision making, it does have extensive competition, and there is undoubtedly some benefit to other techniques. For example, TD(λ) [Sutton, 1988] is a reinforcement technique which is particularly well-suited to situations in which rewards are often delayed; this could at least partially mitigate the problem of a lack of intermediate awards in TMKL models.

One could also envision abandoning reinforcement learning altogether. One option would be to compute a combined expected utility of lower-level decisions when performing a high-level decision. This approach is taken in [Doan et al., 1995], which assumes that probable outcomes and rewards are given, but is presumably also applicable to situations in which those characteristics are learned. For example, when a method for high-level tasks is chosen, it might be useful to consider not only experience of the utility of the methods for the current task but also experience of the utilities of methods for the subtasks of those methods. Another alternative would be the use of a localized search process. Such an approach would require that there be some sort of information available which could allow the reasoning shell to rule out some future decisions and thus identify "dead ends" before it reaches them. The existing TMKL formalism doesn't provide specific knowledge of this sort. It is possible to learn information of this sort through experience [Korf, 1990, Koenig and Simmons, 1995, Furcy and Koenig, 2000]. However, these approaches perform much better if prior estimates of the benefits of different sorts of actions are available. A search-based variant of the REM execution process would be much more effective if TMKL were extended to allow explicit information about expected benefits of actions. As with the above ideas for reinforcement learning, however, relying on additional information of this sort would require an account of where that additional information came from.

9.1.4 Adaptation

There are an enormous variety of potential future projects in the area of adaptation. Many of these projects involve the development of new adaptation strategies; this potential future work is described in Section 9.2. This section focuses on potential improvements to the existing adaptation strategies.

Recall that the situated learning strategy presented in Section 6.1 creates a method for an agent which makes all possible actions on all possible objects available at every step of reasoning (so that selection of actions is done only during execution, via reinforcement learning). The experiments in Chapter 7 show that this strategy produces extremely slow agents for all but the simplest of problems. On the other hand, this strategy is extremely broad in its

applicability; any problem which can be solved by any combination of the known action tasks operating on known values can be solved by this strategy. Note that this does not make the set of problems which can be solved by this strategy a superset of the set which can be solved by the other strategies; for example, the model-transfer strategies involve some knowledge manipulation tasks which involve the production of new values and thus can address some problems which cannot be addressed simply by performing actions on existing values. Trying to address these issues in the situated learning mechanism is probably unproductive; the range of things that it is already able to do is so large that it is usually prohibitively expensive. Future work on the situated learning mechanism would probably do better focusing on reducing the number of alternatives available. For example, the process could filter out combinations of objects and actions which are never possible. In addition, because this strategy defers all choices to the decision making module, the potential enhancements to that module described in Section 9.1.3 would presumably lead to a particularly significant improvement in the agents produced by this strategy.

There are many opportunities to improve the generative planning adaptation strategy described in Section 6.2. One of the major limitations of this strategy is that it can only use tasks whose preconditions and postconditions can be translated into Graphplan's operator language. If a problem requires the use of some other task, it cannot be solved by this strategy. Furthermore, while Graphplan is a very fast planning system, it does have a particularly restrictive operator language. One possibility would be to allow the generative planning mechanism to invoke a variety of planning algorithms depending on the demands of the available tasks. For example, if some of the existing tasks involved universal quantification then UCPOP [Penberthy and Weld, 1992] could be used. There are many issues for future research relating to how to combine and select among a set of planning algorithms.

The relation mapping strategy described in Section 6.3 also presents several opportunities for future work. The current strategy does not have an account of how to select only one relation from a set of relations which link the effects of the existing task to the effects of the new task. Consequently, the strategy just chooses the first one that it finds. In the ADDAM assembly example, this is not an issue because there is only one such relation: inverse-of. It seems reasonable to expect that in many situations there is a single clearly defined relationship between the existing task and the new task. However, there are certainly other situations in which more than one relation is relevant. It would be useful to have some mechanism to decide on one relation to map. It would also be useful to be able to determine if the attempt to map a relation has failed and to try an alternative relation. Lastly, it may be productive to consider a variation of relation mapping that uses more than one relation.

The fixed value production strategy described in Section 6.4.2 also provides some potential for enhancement. One major issue with this strategy is that it is frequently unclear whether and how far the production of a value should be generalized. For example, in one SIRRINE meeting scheduling example, the agent is adapted so that it adds a time slot which runs from 4:30PM until 6PM on Tuesdays. The fixed value production strategy performs a modification which always produces that one particular time slot for meetings of that length when no other slot is available. However, it might be productive instead to generalize this result to always allow meetings until 6PM on Tuesdays, or to always allow meetings until 6PM on all days, etc. For an agent to determine which of these generalizations was appropriate, it would need to have experience with more than one example; the fixed value production strategy only reasons about one example at a time and thus would not be able to address this issue. On the other hand, it may be possible to accomplish something like this by combining the fixed value production strategy with the relation mapping strategy; if the model of the domain had a generalization relation then it might be possible for relation mapping to add steps to the model which impose this generalization. Recall that steps added by relation mapping are made optional and their use is determined by reinforcement learning. Thus if a particular generalization seemed to regularly produce acceptable results, the reinforcement learning would eventually include it; similarly if a generalization went too far, the reinforcement learning would eventually omit it. The most obvious reason why this cannot be done in the existing REM system is the fact that relation mapping is only used for proactive adaptation. In principle, however, it seems like it could be updated to support failure-driven adaptation of this sort.

9.2 Additional Capabilities

The most obvious opportunities to add new capabilities to REM and SIRRINE involve the creation of new adaptation strategies. There is an essentially infinite variety of new ways to adapt a model which have not been thought of yet. Furthermore, there are many existing theories and techniques which may be applicable to the problem of adapting a TMK model. One potential source of new strategies is the fact that adaptation of models is a form of analogical transfer. The use of general purpose analogical transfer mechanisms such as SME [Gentner, 1983] for transfer among TMK models would be an interesting research topic.

Another opportunity to extend this theory would involve merging the set of adaptation strategies in REM, SIRRINE, and Autognostic into a single system. The adaptation strategies in REM are nearly a superset of those in SIRRINE; however, recall (from Section 6.4.2) that SIRRINE has two different variations of the fixed value production strategy. The variation which is not included in REM could certainly be added. More significantly, there are a large number of adaptation strategies in Autognostic that are not in REM or SIRRINE. A system which had access to the adaptation strategies in all three of these systems would have obvious practical benefits in that a particular adaptation challenge that arose would be more likely to be addressed if all of the existing TMK-based adaptation strategies were available. There would also be interesting theoretical issues involved in combining and selecting among this variety of strategies.

Yet another source of additional adaptation strategies could be work in Software Engineering on Design Patterns [Gamma et al., 1994]. This work focuses on abstract descriptions of certain ways of accomplishing common sorts of functionality within Object-Oriented (OO) software. There is a certain degree of compatibility between TMKL and OO software design, as described in Section 8.4. Thus there may be opportunities for applying these Design Patterns to TMKL adaptation. In particular, a key use of Design Patterns is the facilitation of certain kinds of adaptation; in existing work, that adaptation has been manually performed by a software engineer. A first cut at using Design Patterns for automated adaptation of TMK models would be to have certain aspects of the models be annotated by the model designer as filling certain roles in a specified pattern; as needed, the reasoning shell could then apply the kinds of adaptations which are common for that pattern. A more elaborate use would be for the adaptation strategies to automatically detect the presence of these patterns (so that no explicit annotation would be required). An even more elaborate use would be for the adaptation strategies to detect when a portion of the agent involves functionality which *could* be accomplished by a Design Pattern and then first transforms that portion of the agent so that it does fit into the pattern and then performs whatever adaptation is enabled by that pattern as needed.

In addition to new adaptation strategies, there are a variety of other capabilities which could improve this work. One idea for future work is the use of models of external agents. Most agents operate in the service of a particular user. That user may also be using other software agents. Furthermore, the user is often collaborating with other users who also have software agents. In addition, in some domains the user may also be competing with other users who have software agents of their own. The effectiveness of a particular software agent may depend on how well it understands its user, other users, and other software agents. It seems like it would be productive for an agent to have a coherent modeling framework for modeling not only itself (as in the research presented in this dissertation) but also other users and software agents. TMKL may be partially suited to this task. However, additional modeling capabilities may be needed to represent processes which are outside of the control of the agent and may be only partially understood. Future work could consider what can be done with models of external agents and what language features are needed to support those uses.

There are also many kinds of agents which have not been considered in this work. For example, there is an increasingly large variety of computer systems which are embedded in everyday devices such as cellular phones, household appliances, etc. Because these systems are often directly involved in a real, complex, dynamic environment, they seem to pose even stronger demands for the ability to adapt than most agents on a traditional desktop computer. The techniques presented in this dissertation may be partially applicable to this domain as they are. Certainly embedded software agents can perform tasks, use methods, and contain knowledge so TMK-based adaptation techniques may well be appropriate for them. However, there may be additional capabilities which are particularly appropriate for adaptation of embedded systems. For example, an embedded agent may be able to use a model of its physical hardware in addition to using models of the sort presented in this dissertation. In addition, models of external agents (as discussed in the previous paragraph) may be particularly useful for embedded devices since these devices often interact directly with a variety of agents.

Lastly, there is much future work to be done on this theory in the area of evaluation. The results in Chapter 7 do demonstrate that this work provides significant capabilities for a variety of domains and problems. However, the range of challenges for which this work has been tested is clearly a very small fraction of the enormous variety of domains and problems for which adaptation of agents may be useful. A large-scale evaluation of this theory would require a reasoning shell which could be used by many different people in many different situations. Producing a version of REM or SIRRINE which could be used for actual commercial development of software agents would be an enormous task. Such a system would probably need interactive tools for constructing models; SIRRINE has a graphical user interface, but it only provides capabilities for building and displaying models, not constructing them. A commercial version of these tools would also need many abilities for debugging and testing agents. The failure monitoring described in Section 5.3 addresses issues in which a syntactically valid model encounters an invalid

Input	Theory	System	Examples
Model	What the agent does and how it does it	TMKL	ADDAM, web browser
Task	What an agent or piece of an agent does	input and output parameters, given and makes conditions, reference to implementation	Disassemble, Communicate with WWW Server
Method	How an agent or piece of an agent works	provided and additional-result conditions, state-transition machine	Hierarchical Plan Execution, External Display
Concepts	Kinds of knowledge	Slots and values (Loom)	Physical Object, URL
Relations	Kinds of connections between knowledge	Sets of tuples (Loom)	Connects, Document at Location
Input Values	A particular set of inputs for a task	Bindings between parameters and values	{(input-url, "http://thesis.murdocks.org/thesis.pdf")}
Feedback	A logical assertion about the results of the task, provided by the user, in order to guide adaptation	Can be any assertion, but only should-transform assertions are used	should-transform over the Process URL task + a state in which a PDF is the document type + a state in which "acroread ~a" is the chosen command

Table 57: The various kinds of inputs provided to the reflective reasoning processes described in this dissertation.

situation. However, neither SIRRINE nor REM provide any support for detecting and repairing syntax errors in a model. Furthermore, for this work to gain wide-spread acceptance as a commercial tool, it would almost certainly need to address a broader range of languages than just LISP. It is possible to access non-LISP routines in both SIRRINE and REM through the use of foreign function calls (i.e., the model points to a piece of LISP code and that LISP code invokes code in some other language). However, this is a clumsy solution. It may be productive for future work on TMK, particularly work intended to be relevant to a wide range of users, to consider focusing on one or more languages that are more widely accepted for practical software development; examples of such languages include Java, C, and C++.

9.3 Applicability

A key claim of this dissertation is that the representations and algorithms presented here have broad applicability. The variety of example agents presented in Chapter 4 and the experiments with those agents presented in Section 7.1 provide evidence to support this claim. The theoretical analysis in 7.2 provides further evidence for this claim by explicitly proving to what extent and along which dimensions this work is scalable. However, there certainly are limits to the applicability of this research. The discussion of future research in the earlier sections of this chapter state some of these limitations in the context of discussing how those limitations could be overcome. This section expands upon these descriptions to provide a general overview of the applicability of this research.

Table 57 presents an overview of the kinds of inputs which can be provided to the reasoning processes described in this dissertation. As stated in Chapter 1, these processes have three major classes of inputs: models, input values, and feedback. The overall effect of the process is to perform a task within the model given the specified values in the input. Feedback is provided by the user as needed (i.e., when the execution of the model has not produced the desired result). As stated in Chapter 2, models are further decomposed into tasks and methods plus concepts and relations. In the web browsing domain, an example of a model is one which describes a web browser, an example of a task is accessing a document, and an example of an input value is a particular URL for the document to access. An example of feedback which could be provided in this domain would be the fact that a particular external command should have been produced.

There are three major technical products of this research: the TMKL language, the execution algorithms, and the adaptation algorithms. Each of these products is only significant in the context of the other. Without the adaptation strategies, TMKL and the execution algorithms would merely be just another programming language and interpreter; the adaptation algorithms have been the theoretical focus of this work, but those algorithms could not exist without something to adapt and would not be particularly useful without something to do with the adapted results. Below are descriptions of the applicability conditions of these three products. The combinations of these conditions provides the overall applicability of the theory and systems presented in this dissertation.

Applicability of TMKL: Because primitive tasks in TMKL can contain arbitrary LISP code, it is trivially true that any computation which can be performed in LISP can be performed by some TMKL agent. However, there are limits to what aspects of an agent can be explicitly modeled in TMKL. In particular, TMKL is suited to representing the following sorts of characteristics of agents:

- Structure: One key aspect of TMKL models is their representation of the structure of an agent. The implementations of primitive tasks and the concepts are both structural elements of the model: they encode the basic components of the agent. The relations between the concepts, the parameters which link concepts to primitive tasks, and the methods over primitive tasks are also structural elements: they encode the connections over the components. Together these portions of the model provide a description of how the agent is composed.
- Behavior: TMKL models also encode the behaviors of agents. The states and transitions within a method indicate a flow of control; the combination of the different state-transition machines in the methods of TMKL provides an overall view of how an agent works.
- Function: Lastly, TMKL models represent the intended effects of agents. The input and given slots in a task indicate the starting condition for that task and the output and makes slot indicate the ending condition; together these slots provide a description of what the task does. This functional description is an abstraction of the agent. As noted in Section 9.1.1, however, it would be possible to encode other, non-functional abstractions; examples include speed, reliability, security, etc. TMKL does not provide explicit representations of these characteristics and is thus not generally applicable to model-based reasoning problems which emphasize these issues.

Applicability of execution algorithms: The execution mechanism described in Chapter 5 is very broad in its applicability. There are some limits, however, to the applicability of some of the auxiliary mechanisms that support the execution process:

- Symbolic decision making: It is expected that most decisions (selection of methods or transitions) in most TMKL models are performed through purely symbolic analysis (as described in Section 5.2.1). The conditions which can be specified for selecting methods or transitions symbolically are specified in the Loom formalism. Because this is a broad and powerful formalism, the symbolic decision mechanism making is widely applicable. This approach is, however, limited to those decisions for which a complete specification of when a particular selection is appropriate.
- Reinforcement Learning: When the symbolic conditions for more than one of the methods or transitions are met, REM uses reinforcement learning to select among those options. Reinforcement learning can be applied to any task in which a known set of actions can produce observable rewards based on observable states. However, reinforcement learning may require extensive experience before it develops an effective policy. Thus this mechanism is not applicable to problems where effective solutions are needed immediately. Consequently, REM is only applicable to agents in which nearly all of the execution decisions are specified symbolically *or* it is acceptable for an agent to run many times before consistently producing adequate results.
- Failure monitoring: The failure monitoring techniques described in Section 5.3 cover the different portions of REM's execution mechanism for which a syntactically valid TMKL model can fail during execution (e.g., when a task has no applicable methods). There are two major limits to the applicability of this mechanism. First, it cannot detect syntactic errors; if such an error appears occurs, the result is likely to be a crash within REM. Second, it can only detect incorrect behavior when the model explicitly states that the behavior is incorrect. For example, if the makes condition for a task is underspecified, the model will operate fine under normal conditions, but if the agent encounters a condition in which the results are not adequate but also not explicitly excluded by the makes condition, then failure monitoring will miss an opportunity to detect a problem. In that case, feedback must be provided by the user or the need for adaptation will be missed entirely.

Applicability of adaptation algorithms: The adaptation algorithms presented in Chapter 6 are shown in Chapter 7 to address a variety of different adaptation challenges in a variety of domains. However, these algorithms certainly do not have an unlimited range of use. The algorithms for adaptation have the following conditions for applicability:

- Situated Learning: The situated learning learning algorithm has extremely broad applicability: any task which can be performed using the known actions on known values can be performed with this algorithm. The main drawback to this algorithm is that it is extremely slow except for very small problems; because any possible sequence of actions may be tried, it can take a very long time to find one which accomplishes the desired result. Another major limitation of this algorithm is the fact that it cannot address tasks which require computation or observation (except in observing whether the desired result has been met); this is because all combinations of actions and values are enumerated before execution so any new values encountered during execution cannot be acted upon. Furthermore, because this mechanism involves trying out actions without knowing if they will work, it is not applicable to domains in which some incorrect actions cause catastrophic failure.

- Generative Planning: The generative planning adaptation algorithm is slightly less broad in its applicability than situated learning. In particular, an agent using generative planning needs to know not only what the available actions are but also what the effects of those actions are; in contrast, the situated learning agent does not need a specification of the effects because it simply executes the actions and observes the results. Note that for a given planner, there are many kinds of actions whose effects cannot be represented in the planner's formalism. For example, REM uses Graphplan, which cannot represent actions which are non-deterministic or universally quantified, etc. Thus the generative planning adaptation mechanism in REM is only applicable to tasks which can be addressed without any actions of those sorts. In addition, generative planning, like situated learning, is often very expensive.

- Proactive Model Transfer via Relation Mapping: Proactive model transfer is, in principle, an extremely broad area of research. This dissertation has described only one mechanism for proactive transfer; that mechanism is relation mapping. Relation mapping is applicable to desired tasks for which there exists a known task such that a single relation directly maps the intended results of the known task to the intended results of the desired task *and* the known process can be adapted to suit the desired task by applying the relation to some of the knowledge produced by the agent. Note that it is often not known in advance which knowledge needs to be mapped; this is why the relation mapping algorithm imposes optional transitions and uses reinforcement learning to resolve the options. Unlike the situated learning algorithm, reinforcement learning is isolated to only a few areas; this typically avoids the extreme cost that the situated learning imposes (as shown in Section 7.1). However, like situated learning, this mechanism does perform some trial and error reasoning and thus is ill-suited to domains in which incorrect actions can lead to catastrophic failures.

- Failure-Driven Model Transfer via Fixed Value Production: Like proactive model transfer, failure-driven model transfer is a broad topic of which this work has only addressed a small part. The fixed value production mechanism is applicable to adaptation problems which can be addressed by having a single specific task always output a single fixed value under the same knowledge conditions that were encountered in the example. It cannot cause any additional action or computation to be performed (except when producing a value at one point causes additional action or computation later in the model). In addition, this mechanism is the only one which uses feedback from the user, and it only uses one kind of feedback: a single should-transform assertion. There is a virtually limitless range of other kinds of feedback which could be useful for adaptation; see Section 9.1.2 for more on this subject.

- Failure-Driven Model Transfer via Generative Planning Repair: The primary requirement for this adaptation strategy is that a task was encountered for which no method was applicable. The mechanism builds a new method to address this failure. Beyond that, all of the applicability conditions for the main generative planning adaptation strategy also apply to this strategy, since they both use the same planning process.

Note that overall applicability of adaptation in REM is not merely limited to the union of the capabilities above. Because REM may execute an agent many times, different adaptations to the agent can accumulate resulting in an agent which could not have been produced from the original agent by any one strategy alone. Thus the overall applicability of adaptation in REM extends to problems which cannot be addressed by any single invocation of a single strategy but which can be addressed by a combination of adaptation strategies. For example, the work on

cumulative adaptation described in Section 7.1.3 shows how proactive transfer can be used to adapt a model which includes steps produced by generative planning repair.

The execution and adaptation algorithms along with the models that these algorithms use constitute a theory of reflection and self-adaptation. This theory has been demonstrated to be effective for a wide range of problems, particularly ones which demand changes in functionality in complex, knowledge intensive environments. Future work may involve not only further improving the effectiveness of this theory on the sorts of problems for which it is already useful but also extending its range of applicability.

Chapter 10

Conclusions

Reflective model-based adaptation is a reasoning technique whereby an agent uses a model of itself to learn new capabilities or augment existing capabilities. The work on this topic presented in this dissertation leads to several theoretical contributions, claims, and conjectures. Below is a brief list of topics for which this research has generated the most substantial conclusions; after that, the conclusions regarding these topics are described in detail:

- Agent Modeling
- Analogical Reasoning
- Machine Learning
- Planning
- Meta-Reasoning and Reflection

Agent Modeling: This research has produced several substantial claims regarding modeling of agents. Task-Method-Knowledge (TMK) models enable reasoning at the level of the overall architecture of an agent. Reasoning at this level is particularly crucial for modifications to an agent which affect its overall function, such as changing a disassembly agent into an assembly agent. TMK models are teleological in nature; i.e., they provide explicit connections between the functions of an agent and the ways in which those functions can be accomplished. The model-based adaptation techniques developed in this research use this connection to derive ways to accomplish new functions from the ways that are used to accomplish existing functions.

One major contribution of this research is TMKL, a new formalism for representing TMK models. The most notable novel characteristic of TMKL is its representation of complex logical expressions in tasks and methods, as described in Chapter 2. These expressions are a crucial element in the level of depth to which a model can represent what effects some portion of a process has. This information is particularly valuable in adaptation before execution. In this situation, reasoning about tasks and methods needs to be based on abstract knowledge about the kinds of concepts and relations that they affect since there is no trace to provide concrete information about specific instances.

The relation mapping algorithm in Chapter 6 provides a particularly compelling example of how information about the relations affected by a task can be used to guide adaptation.

Of course, any reasoning based on models is only useful if models are available. The existence of the models described in Chapter 4 provides a proof-of-concept for the availability of TMKL models. However, it does not conclusively prove that models of this sort are likely to be produced by a typical agent designer for a typical agent given a reasonable amount of effort. However, we do have some evidence that this is the case, i.e., that the knowledge requirements for model-based adaptation are reasonable. The core content of a Task-Method-Knowledge model involves information about what the agent is doing and how it is doing it. Clearly someone building an agent must have this information at the time it is being built. It is not unreasonable to conjecture, therefore, that the builder can potentially articulate this information using a particular formalism (in this case, TMKL). Alternatively, it is also reasonable to expect that an analyst looking at the code and/or design documentation for a system can extract its purpose and composition. There is preliminary evidence to validate these conjectures from the efforts in combining model-based adaptation with Software Engineering techniques and formalisms, e.g., the work done in the MORALE [Abowd et al., 1997] project.

Analogical Reasoning: This work also makes a variety of contributions to analogical reasoning. The most significant of these contributions involves the kinds of information that is transfered from an old case to a new one. The agents developed here do not adapt the results of their reasoning processes; instead they adapt the reasoning processes themselves. Unlike traditional analogical systems, what is being adapted is the entire design of an agent, not just a set of facts and/or actions that constitute a problem solving or planning case. This distinction is particularly relevant in domains for which similar processes can produce radically different results. In these environments, it may be possible to adapt a process even when the results are simply too dissimilar for transfer.

Derivational analogy, which also performs transfer over processes, uses knowledge of traces, as does the failure-driven transfer algorithm presented in Chapter 6. The greatest difference between the uses of traces in this research and the use of traces in derivational analogy is that the traces in this work are directly connected to the model (as described in Chapter 3). Thus this work combines knowledge of the events that occured in a particular reasoning episode with broader knowledge of the kinds of events that occur across episodes.

There are also significant differences in the kind of traces used here and the way that they are produced as compared with existing approaches. SIRRINE and REM automatically produce traces during reasoning by keeping track of what tasks, methods, and knowledge were employed at each step of execution process. Because the reasoning shells perform the execution using the model, no extra effort is required to identify the pieces of the processes being executed; the reasoning shells simply need to record information that they must have anyway to do the execution. The trace can only be produced when a model is available (since it is based on the execution of the model); however, since the model is available the production of the trace is automatic. Furthermore, the production of the trace does not impose an overwhelming additional expense over what the costs would be without the trace. The analysis in Section 7.2.1 shows there are only moderate extra costs (above the uncontrollable costs imposed by primitive actions and logical expressions) incurred by the entire execution process including trace generation and monitoring. More specifically, in the worst case, these costs are linear in the total size of the trace produced during the execution, under some reasonable assumptions about the organization of the model. Even when these assumptions about the model are removed, the costs are still, in the worst case, only proportional to the product of the size of the trace and the size of the model.

Furthermore, the costs of using this trace are also reasonable. One of the techniques which uses a trace is the failure-driven model transfer process using fixed value production. The worst case performance of this technique is linear in the size of the trace but may also be affected by the size of the model, the amount of feedback provided, and the number of failures which were detected (as detailed in Section 7.2.2.3). The other technique which benefits from a trace is failure-driven model transfer using generative planning; this process is shown (in Section 7.2.2.4) to have worst case performance which is comparable to the sum of the worst case performance for the fixed value production cost and the performance of the generative planner. This cost can certainly be much more expensive than fixed value production (since planning can be very expensive); however, the results in Section 7.1.3 show that for some complex problems it can, at least, be substantially less than the cost of planning the entire process.

Note the contrast between the demand that the failure-driven model transfer technique developed here makes for a *failed* trace and the demand that derivational analogy makes for a *successful* trace. The existence of a failed trace is a very reasonable requirement for adaptation because the system automatically produces it during the failed reasoning episode which is providing the motivation for adaptation. The existence of a successful reasoning trace is much less reasonable; if the system does not already know how to do something, it is unreasonable to expect that the

user will provide every step of a reasoning process which does that thing. It is possible to base derivational analogy on successful traces of past reasoning episodes for similar problems. This approach can be effective if an agent first encounters many simple problems which can be efficiently solved and then later encounters similar but harder problems, thus allowing the reuse of the derivations of the simple problems. However, if an agent is not guaranteed to have simple problems arise first, it is unreasonable to suspect that it should have successful traces available for the hard problems it encounters. Thus it can be concluded that the knowledge requirements of model-based adaptation with respect to traces are thus more generally reasonable than those of derivational analogy.

Machine Learning: Model-based adaptation is a form of machine learning; adaptation allows an agent to perform a task which it was not already capable of performing. Most traditional machine learning approaches involve selection of an alternative from a fixed set of options; those options may be actions, categories, etc. Model-based adaptation allows the development of new ways of reasoning for new kinds of problems. This differs substantially from typical machine learning techniques; addressing this different issue is a significant contribution to the machine learning field.

The work presented in this dissertation also makes more specific contributions to a particular sub-topic within machine learning: reinforcement learning. Reinforcement learning occurs within REM in the selection of methods and selection of subtasks within methods (as described in Chapter 5). Unlike conventional applications of reinforcement learning, however, the quantity of decisions that need to be learned through reinforcement is constrained by a model; i.e., the model directly addresses some or all of the choices that need to be made during execution, leaving a relatively small number to be solved by reinforcement learning. Execution of models using reinforcement learning for isolated locations provides significant advantages over reasoning by reinforcement learning alone. It is well known that reinforcement learning (like many similar approaches) has a great deal of trouble with problems which have very large state spaces and many goal conflicts. This limitation of reinforcement learning is a simple reflection of the fact that conflicting goals tend to suggest opportunities which appear to be productive but, in reality, are not. Sifting through all of those unproductive false leads through experience can be extremely time consuming. The results presented in Chapter 7 lead to the conclusion that for problems involving many conflicting goals, models can enable an agent to avoid being overwhelmed by these conflicts and thus obtain results much more quickly.

Planning: This research provides two major contributions to the topic of planning: reuse and localization. The reuse contribution involves the fact that the generative planning adaptation strategy (in Chapter 6) does not merely use the plan generated but rather it abstracts that plan into a method which can also be used for similar problems (in particular, on problems which involve performing the same actions on different objects). Because these abstracted plans are stored as models, it is possible for later model-based adaptation to modify them to accomplish different effects as well. Furthermore, the complexity analysis in Section 7.2.2.1 shows that the mechanism does not add any asymptotic cost above the cost of running the external planner (under reasonable assumptions).

The localization contribution to generative planning is derived from the use of generative planning in failure-driven transfer, as described in Section 6.4. In this process, generative planning is used to create a method not for a main task which is provided by a user but rather for an isolated subtask which is identified by the credit-assignment portion of the failure-driven transfer algorithm. Thus some of the reasoning process is transfered from the existing agent. This is particularly beneficial if the overall problem being solved is very complex but a particular subtask which has been diagnosed as responsible for the failure is relatively simple. Furthermore, the results presented in Section 7.1.3 show that localization of generative planning can provide substantial advantages even when the task being addressed by planning is the most complex part of the process; using models to remove even a few minor details from a long planning process can lead to substantial improvement in speed.

Meta-Reasoning and Reflection: There is a very diverse range of systems which reason about reasoning. The reflection techniques in Software Engineering that are in popular use tend to focus on accessing properties of declarative data structures (particularly objects in an object-oriented framework) and not on processing. More advanced research in reflection within Software Engineering does involve processing information, but it tends to consider very low-level sorts of processing events (e.g., procedure invocations) and typically does so with respect to non-functional traits such as efficiency and security. In contrast, the model-based adaptation approach presented here reasons at a variety of relatively high levels of abstraction (because TMKL models are hierarchical and encode the overall functions of hierarchies and subhierarchies) and does so in the support of modification of functional characteristics.

Many meta-reasoning techniques in Artificial Intelligence are also concerned with non-functional characteristics, especially processing time. Meta-reasoning of this sort tends to focus on deciding which of multiple available approaches is likely to be fastest and/or whether it is worth the cost of continuing to reason about an existing adequate solution. These techniques seem complementary to the kinds of reflective reasoning performed in this work, which focuses on the development of new capabilities.

A particularly crucial aspect of any reasoning techniques is the range of problems that it can address; model-based adaptation differs from existing approaches to meta-reasoning along this dimension. Model-based adaptation can attempt to address a problem if the differences between the old task and the new can be mapped into known relations over task-related knowledge (e.g., knowledge of inversion in the disassembly to assembly example) or if the differences can be defined by feedback regarding specific knowledge items for a specific execution (e.g., knowledge of Acrobat reader in the PDF browsing example).

Note that some of problems of this sort may be successfully addressed directly by model-based techniques alone. However, problems which involve complex transformations of complex elements (e.g., some of the transformations in the disassembly to assembly example in Section 6.3) require additional reasoning techniques such as reinforcement learning or generative planning. These additional techniques do impose additional computational costs; since they are known to be overwhelmingly expensive for very complex problems, this is a substantial concern. Fortunately, the results presented in Section 7.1 demonstrate that models can be used to isolate relatively small portions of a reasoning process which require additional reasoning. The computational analysis in Section 7.2 further expands upon these results. The relation mapping algorithm produces a model which may have optional behaviors which must be resolved through reinforcement learning. A key benefit of this algorithm is that the cost of model adaptation depends only on the model and not on the particular problem being solved. Specifically, the worst case performance of relation mapping quadratic in the size of the model under certain assumptions about the organization of the model and cubic in the size of the model if those assumptions are removed; in either case, this performance is independent of the particular set of inputs provided. When a very challenging input is provided to a task (e.g., the roof design discussed in Section 7.1.1 with five or more boards), this input does not affect the adaptation cost, leading to very steady performance. By restricting powerful but potentially very expensive techniques to those locations where they are absolutely essential, model-based adaptation is able to avoid the potentially enormous expense of using these techniques in isolation.

A final contribution of this research is a more general observation about levels of abstraction in reflection and learning. The models described here represent an agent's reasoning about itself at the level of its overall design. This design information includes not only specific atomic computations and actions that the agent performs but also the way that these primitive elements are combined to form an overall behavior which accomplishes an overall function. This research establishes the conclusion that reflection at the level of design is an effective approach for adapting to new functional requirements.

Appendix

The Web Browser TMKL model

This is the complete TMKL model of the web browsing agent which is described in Section 4.2. It is presented here in order to provide a comprehensive example of TMKL in use. Note that semi-colons denote comments within the model.

```
(defcontext web-browsing
  :theory (tmkl))

(in-context web-browsing)

(defrelation text
  :range string
  :characteristics :single-valued)

(defrelation text-matches
  :is (:satisfies (?x ?y) (:same-as (text ?x) (text ?y))))

; Concept: location
; A location is any knowledge item which can act as a
;  description for where a piece of data is located (e.g., a
;  URL).
(defconcept location)

; Concept: computer
; A computer may either be a local terminal where
;  information is being stored or an external machine such as
;  a web server.
(defconcept computer :is-primitive location)

; Concept: url
; A URL is simply a string such as "http://www.foo.com/smurf.html"
(defconcept URL :is-primitive location :roles (text))

; Concept: document
; A document is a big chunk of information which comes back
;  from a server.  It includes both a tag which identifies
;  the type of the data and a piece of unformatted data.
(defconcept document :roles (document>tag document>data))

(defrelation document>tag
  :domain document
  :range tag
  :characteristics :single-valued)

(defrelation document>data
  :domain document
  :range data
  :characteristics :single-valued)

; Concept: tag
```

```
; A tag is simply a string corresponding to a file type
;  (e.g. "text/html", "application/postscript", "video/mpeg")
(defconcept tag :roles (text))

; Concept: data
; Data is simply a sequence of bits retrieved from the
;  server, e.g. a postscript file.
(defconcept data :roles (data>filename))

(defrelation data>filename
  :domain data
  :range filename
  :characteristics :single-valued)

; Concept: filename
; A filename is a string identifying a file, e.g. "foo.ps"
(defconcept filename :roles (text))

; Concept: command
; A command is simply a format string corresponding to an
;  action by the server.  For example, the abstract ghostview
;  command is "ghostview ~a".  A concrete instance of this
;  abstract command for the file foo.ps", would be "ghostview
;  foo.ps".
(defconcept command :roles (text))
(defconcept abstract-command :is-primitive command)
(defconcept concrete-command :is-primitive command)

; These are some additional relations which describe abstract
;  properties that exist across different concepts.

(defrelation document-at-location
  :domain document
  :range location)

(defrelation command-displays-document
  :domain command
  :range document)

(defrelation executed
  :domain computer
  :range concrete-command)

(tell (external-state-relation executed)
      (external-state-relation document-at-location))

(defcontext tim :theory
  (tmkl web-browsing) :open-closed-mode :closed)
(in-context tim)

(tell (:create local-host computer))

(defconcept internal-display-tag
  :is-primitive tag)

(defrelation tag>command
  :domain tag
  :range command
  :characteristics :single-valued)
```

```
(tell (:about tag-txt
              (:create internal-display-tag)
              (text "text/plain"))
      (:about tag-html
              (:create internal-display-tag)
              (text "text/html"))
      (:about tag-ps
              (:create tag)
              (text "application/postscript"))
      (:about tag-xbm
              (:create tag)
              (text "image/x-xbitmap"))
      (:about tag-jpeg
              (:create tag)
              (text "image/jpeg")))

(tell (:about command-ghostview
              (:create abstract-command)
              (text "ghostview ~a"))
      (:about command-xv
              (:create abstract-command)
              (text "xv ~a")))

(tell (tag>command tag-ps command-ghostview)
      (tag>command tag-xbm command-xv)
      (tag>command tag-jpeg command-xv))

; Parameters

(define-parameter input-url url)
(define-parameter server-reply document)
(define-parameter server-tag tag)
(define-parameter server-data data)
(define-parameter chosen-abstract-command abstract-command)
(define-parameter chosen-concrete-command concrete-command)

; Tasks

(define-task process-url
  :input (input-url)
  :makes (:for-some
          (?doc ?cmd)
          (:and
           (document-at-location ?doc (value input-url))
           (command-displays-document ?cmd ?doc)
           (executed local-host ?cmd)))
  :by-mmethod (process-url-method))

(define-task communicate-with-www-server
  :input (input-url)
  :output (server-reply)
  :makes (:and
          (document-at-location
           (value server-reply) (value input-url))
          (document-at-location
           (value server-reply) local-host))
  :by-mmethod (communicate-with-server-method))
```

```
(define-task request-from-server
  :input (input-url)
  :output (server-reply)
  :by-procedure request-from-server-proc
  :makes (document-at-location
          (value server-reply) (value input-url)))

(define-task receive-from-server
  :input (input-url server-reply)
  :given (document-at-location
          (value server-reply) (value input-url))
  :by-procedure receive-from-server-proc
  :asserts (document-at-location
            (value server-reply) local-host))

(define-task display-file
  :input (server-reply)
  :given (document-at-location
          (value server-reply) local-host)
  :makes (:for-some
          (?cmd)
          (:and
           (command-displays-document ?cmd (value server-reply))
           (executed local-host ?cmd)))
  :by-mmethod (display-file-method))

(define-task interpret-reply
  :input (server-reply)
  :output (server-tag server-data)
  :given (document-at-location
          (value server-reply) local-host)
  :binds ((server-tag
           ?x (document>tag (value server-reply) ?x))
          (server-data
           ?x (document>data (value server-reply) ?x))))

(define-task display-interpreted-file
  :input (server-reply server-tag server-data)
  :given (:and
          (document>tag
           (value server-reply) (value server-tag))
          (document>data
           (value server-reply) (value server-data)))
  :makes (:for-some
          (?cmd)
          (:and
           (command-displays-document ?cmd (value server-reply))
           (executed local-host ?cmd)))
  :by-mmethod (internal-display external-display))

(define-task execute-internal-display
  :input (server-reply server-tag server-data)
  :given (:and
          (document>tag
           (value server-reply) (value server-tag))
          (document>data
           (value server-reply) (value server-data)))
  :makes (:for-some
          (?cmd)
```

```
          (:and
           (command-displays-document ?cmd (value server-reply))
           (executed local-host ?cmd)))
  :by-mmethod (display-html-internal display-text-internal))

(define-task execute-html-internal-display
  :input (server-reply server-data)
  :given (document>data (value server-reply) (value server-data))
  :makes (:for-some
          (?cmd)
          (:and
           (command-displays-document ?cmd (value server-reply))
           (executed local-host ?cmd)))
  :by-procedure execute-html-internal-display-proc)

(define-task execute-text-internal-display
  :input (server-reply server-data)
  :given (document>data (value server-reply) (value server-data))
  :makes (:for-some
          (?cmd)
          (:and
           (command-displays-document ?cmd (value server-reply))
           (executed local-host ?cmd)))
  :by-procedure execute-text-internal-display-proc)

(define-task select-display-command
  :input (server-tag)
  :output (chosen-abstract-command)
  :binds ((chosen-abstract-command
           ?x (tag>command (value server-tag) ?x))))

(define-task compile-display-command
  :input (server-reply server-data chosen-abstract-command)
  :output (chosen-concrete-command)
  :given (document>data (value server-reply) (value server-data))
  :by-procedure compile-display-command-proc
  :makes (command-displays-document
          (value chosen-concrete-command)
          (value server-reply)))

(define-task execute-display-command
  :input (chosen-concrete-command)
  :makes (executed local-host
                   (value chosen-concrete-command))
  :by-procedure execute-display-command-proc)

; Methods

(define-mmethod process-url-method
  :series (communicate-with-www-server display-file))

(define-mmethod communicate-with-server-method
  :series (request-from-server receive-from-server))

(define-mmethod display-file-method
  :series (interpret-reply display-interpreted-file))

(define-mmethod internal-display
  :provided (internal-display-tag (value server-tag))
```

```
  :series (execute-internal-display))

(define-mmethod display-html-internal
  :series (execute-html-internal-display))

(define-mmethod display-text-internal
  :series (execute-text-internal-display))

(define-mmethod external-display
  :provided (:not (internal-display-tag (value server-tag)))
  :series (select-display-command compile-display-command
                                  execute-display-command))
```

Bibliography

[Abowd et al., 1997] Abowd, G., Goel, A. K., Jerding, D. F., McCracken, M., Moore, M., Murdock, J. W., Potts, C., Rugaber, S., and Wills, L. (1997). MORALE – Mission oriented architectural legacy evolution. In *Proceedings International Conference on Software Maintenance 97*, Bari, Italy.

[Albert and Aha, 1991] Albert, M. K. and Aha, D. W. (1991). Analyses of instance-based learning algorithms. In *Proceedings of the Ninth National Conference on Artificial Intelligence - AAAI-91*, pages 553–558, Anaheim, CA. AAAI Press.

[Allemang, 1997] Allemang, D. (1997). ZD tutorial. Unpublished.

[Anderson, 1983] Anderson, J. (1983). A spreading activation theory of memory. *Journal of Verbal Learning and Verbal Behavior*, 22.

[Bhatta, 1995] Bhatta, S. R. (1995). *Model-Based Analogy in Innovative Device Design*. PhD dissertation, Georgia Institute of Technology, College of Computing.

[Blum and Furst, 1997] Blum, A. and Furst, M. L. (1997). Fast planning through planning graph analysis. *Aritificial Intelligence*, 90:281–300.

[Boddy and Dean, 1989] Boddy, M. and Dean, T. (1989). Solving time-dependent planning problems. In Sridharan, N. S., editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence - IJCAI-89*, pages 979–984, Detroit, MI, USA. Morgan Kaufmann.

[Brazier et al., 1997] Brazier, F., Dunin Keplicz, B., Jennings, N., and Treur, J. (1997). DESIRE: Modelling multi-agent systems in a compositional formal framework. *International Journal of Cooperative Information Systems*, 6:67–94. Special Issue on Formal Methods in Cooperative Information Systems: Multi-Agent Systems.

[Brill, 1993] Brill, D. (1993). Loom reference manual. http://www.isi.edu/isd/LOOM/documentation/manual/quickguide.html. Accessed August 1999.

[Cazzola et al., 1999] Cazzola, W., Sosio, A., and Tisato, F. (1999). Reflection and object-oriented analysis. In Cazzola, W., Stroud, R. J., and Tisato, F., editors, *Proceedings of the First Workshop on Object-Oriented Reflection and Software Engineering - OORaSE-99*, pages 95–106. University of Milano Bicocca.

[Chandrasekaran, 1988] Chandrasekaran, B. (1988). Generic tasks as building blocks for knowledge-based systems: The diagnosis and routine design examples. *Knowledge Engineering Review*, 3(3):183–219.

[Clements, 1996] Clements, P. C. (1996). A survey of architecture description languages. In *Eighth International Workshop on Software Specification and Design*, Germany.

[Cover and Hart, 1967] Cover, T. and Hart, P. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):12–27.

[Cox, 1996] Cox, M. T. (1996). *Introspective multistrategy learning: Constructing a learning strategy under reasoning failure*. PhD dissertation, Georgia Institute of Technology, College of Computing.

[Davies, 1999] Davies, J. R. (1999). An evaluation of SIRRINE2 as a cognitive architecture based on a model of human arithmetic. http://www.cc.gatech.edu/~jimmyd/research/despina/abstract.html. Accessed January 2001.

[Davis, 1979] Davis, R. (1979). Interactive transfer of expertise: Acquisition of new inference rules. *Artificial Intelligence*, 12:121–157.

[Davis, 1984] Davis, R. (1984). Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410.

[de Kleer and Brown, 1984] de Kleer, J. and Brown, J. S. (1984). A qualitative physics based on confluences. *Artificial Intelligence*, 24:7–83.

[Dent et al., 1992] Dent, L., Boticario, J., Mitchell, T., Sabowski, D., and McDermott, J. (1992). A personal learning apprentice. In Swartout, W., editor, *Proceedings of the Tenth National Conference on Artificial Intelligence - AAAI-92*, pages 96–103, San Jose, CA. MIT Press.

[Doan et al., 1995] Doan, A., Haddawy, P., and Charles E. Kahn, J. (1995). Decision-theoretic refinement planning: A new method for clinical decision analysis. In *Proceedings of the Nineteenth Annual Symposium on Computer Applications in Medical Care - SCAMC-95*.

[Fischer, 2001] Fischer, G. (2001). User modeling in human-computer interaction. *User Modeling and User-Adapted Interaction (UMUAI)*. Contribution to the Tenth Anniversary Issue; In Press.

[Freed et al., 1992] Freed, M., Krulwich, B., Birnbaum, L., and Collins, G. (1992). Reasoning about performance intentions. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pages 7–12.

[Furcy and Koenig, 2000] Furcy, D. and Koenig, S. (2000). Speeding up the convergence of real-time search. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence - AAAI-00*, Austin, TX.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley Longman, Inc.

[Garlan et al., 1997] Garlan, D., Monroe, R. T., and Wile, D. (1997). Acme: An architecture description interchange language. In *Proceedings of the Seventh Annual IBM Centre for Advanced Studies Conference - CASCON-97*, pages 169–183, Toronto, Ontario.

[Genesereth, 1983] Genesereth, M. R. (1983). An overview of meta level architectures. In *Proceedings of the Third National Conference on Artificial Intelligence - AAAI-83*, pages 119–124, Washington, D.C.

[Gentner, 1983] Gentner, D. (1983). Structure mapping: A theoretical framework for analogy. *Cognitive Science*, 7.

[Goel, 1989] Goel, A. K. (1989). *Integration of Case-Based Reasoning and Model-Based Reasoning for Adaptive Design Problem Solving*. PhD dissertation, The Ohio State University, Dept. of Computer and Information Science.

[Goel et al., 1994] Goel, A. K., Ali, K., Donnellan, M., Gomez, A., and Callantine, T. (1994). Multistrategy adaptive navigational path planning. *IEEE Expert*, 9(6):57–65.

[Goel et al., 1997a] Goel, A. K., Beisher, E., and Rosen, D. (1997a). Adaptive process planning. Poster Session of the Tenth International Symposium on Methodologies for Intelligent Systems.

[Goel et al., 1997b] Goel, A. K., Bhatta, S. R., and Stroulia, E. (1997b). Kritik: An early case-based design system. In Maher, M. L. and Pu, P., editors, *Issues and Applications of Case-Based Reasoning to Design*. Lawrence Erlbaum Associates.

[Goel et al., 1996] Goel, A. K., Gomez, A., Grué, N., Murdock, J. W., Recker, M., and Govindaraj, T. (1996). Explanatory interface in interactive design environments. In Gero, J. S. and Sudweeks, F., editors, *Proceedings of the Fourth International Conference on Artificial Intelligence in Design - AID-94*, pages 387 – 405, Stanford, California. Kluwer Academic Publishers.

[Goel and Murdock, 1996] Goel, A. K. and Murdock, J. W. (1996). Meta-cases: Explaining case-based reasoning. In Smith, I. and Faltings, B., editors, *Proceedings of the Third European Workshop on Case-Based Reasoning - EWCBR-96*, Lausanne, Switzerland. Springer.

[Griffith and Murdock, 1998] Griffith, T. and Murdock, J. W. (1998). The role of reflection in scientific exploration. In *Proceedings of the Twentieth Annual Conference of the Cognitive Science Society*.

[Griffith et al., 1997] Griffith, T., Nersessian, N., Goel, A. K., and Clement, J. (1997). Exploratory problem solving in scientific reasoning. In *Proceedings of the Nineteenth Annual Conference of the Cognitive Science Society*. Lawrence Erlbaum Associates.

[Hammond, 1989] Hammond, K. J. (1989). *Case-Based Planning: Viewing Planning as a Memory Task*. Academic Press.

[Hayes-Roth, 1995] Hayes-Roth, B. (1995). An architecture for adaptive intelligent systems. *Artificial Intelligence*, 72:329–365.

[Hayes-Roth and Hayes-Roth, 1979] Hayes-Roth, B. and Hayes-Roth, F. (1979). A cognitive model of planning. *Cognitive Science*, 3:275–310.

[Kaelbling et al., 1996] Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4.

[Kant, 1781] Kant, I. (1781). *The critique of pure reason*. On-Line Edition: Virginia Polytechnic Institute and State University. translated by J.M.D. Meiklejohn.

[Keuneke and Allemang, 1989] Keuneke, A. and Allemang, D. (1989). Exploring the no function in structure principle. *Journal of Experimental and Theoretical Artificial Intelligence*, 1:79–89.

[Kirlik, 1998] Kirlik, A. (1998). The ecological expert: Acting to create information to guide action. In *Proceedings of the 1998 Conference on Human Interaction with Complex Systems, HICS-98*, Dayton, OH.

[Knoblock, 1994] Knoblock, C. A. (1994). Automatically generating abstractions for planning. *Artificial Intelligence*, 68:243–302.

[Koenig and Simmons, 1995] Koenig, S. and Simmons, R. (1995). Real-time search in non-deterministic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence - IJCAI-95*, pages 1660–1667.

[Korf, 1990] Korf, R. E. (1990). Real-time heuristic search. *Artificial Intelligence*, 42(2–3):189–211.

[Leake et al., 1995] Leake, D. B., Kinley, A., and Wilson, D. (1995). Learning to improve case adaptation by introspective reasoning and cbr. In *Proceedings of the First International Conference on Case-Based Reasoning - ICCBR-95*, Sesimbra, Portugal.

[Liver and Allemang, 1985] Liver, B. and Allemang, D. T. (1985). A functional representation for software reuse and design. *International Journal of Software Engineering and Knowledge Engineering*, 5(2):227–269.

[MacGregor, 1999] MacGregor, R. (1999). Retrospective on Loom. http://www.isi.edu/isd/LOOM/papers/macgregor/Loom_Retrospective.html. Accessed August 1999.

[McCluskey, 1998] McCluskey, G. (1998). Using Java reflection. http://developer.java.sun.com/developer/technicalArticles/ALT/Reflection. Accessed June 2001.

[McDermott and Doyle, 1980] McDermott, D. and Doyle, J. (1980). Non-monotonic logic I. *Artificial Intelligence*, 13(1&2).

[Melis and Ullrich, 1999] Melis, E. and Ullrich, C. (1999). Flexibly interleaving processes. In *Proceedings of the Third International Conference on Case-Based Reasoning - ICCBR-99*, pages 263–275.

[Minsky, 1975] Minsky, M. (1975). A framework for representing knowledge. In Winston, P. H., editor, *The Psychology of Computer Vision*. McGraw-Hill.

[Mitchell, 1997] Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, New York.

[Mitchell et al., 1989] Mitchell, T. M., Allen, J., Chalasani, P., Cheng, J., Etzioni, O., Ringuette, M., and Schlimmer, J. C. (1989). Theo: A framework for self-improving systems. In VanLehn, K., editor, *Architectures for Intelligence*. Erlbaum.

[Mitchell et al., 1986] Mitchell, T. M., Keller, R., and Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80.

[Muñoz-Avila et al., 1999] Muñoz-Avila, H., Aha, D. W., Breslow, L., and Nau, D. (1999). HICAP: An interactive case-based planning architecture and its application to noncombatant evacuation operations. In *Proceedings of the Ninth Conference on Innovative Applications of Artificial Intelligence - IAAI-99*, Orlando, FL. AAAI Press. NCARAI TR AIC-99-002.

[Murdock, 1998a] Murdock, J. W. (1998a). Modeling computation: A comparative synthesis of TMK and ZD. Technical Report GIT-CC-98-13, Georgia Institute of Technology, College of Computing.

[Murdock, 1998b] Murdock, J. W. (1998b). Prolegomena to a task-method-knowledge theory of cognition. In *Proceedings of the Twentieth Annual Conference of the Cognitive Science Society*.

[Murdock and Goel, 1999a] Murdock, J. W. and Goel, A. K. (1999a). An adaptive meeting scheduling agent. In *Proceedings of the First Asia-Pacific Conference on Intelligent Agent Technology - IAT-99*, Hong Kong.

[Murdock and Goel, 1999b] Murdock, J. W. and Goel, A. K. (1999b). SIRRINE2 user's manual. From http://www.cc.gatech.edu/morale/tools/sirrine. Accessed March 2001.

[Murdock and Goel, 1999c] Murdock, J. W. and Goel, A. K. (1999c). Towards adaptive web agents. In *Proceedings of the Fourteenth IEEE International Conference on Automated Software Engineering - ASE-99*, Cocoa Beach, FL.

[Murdock and Goel, 2001] Murdock, J. W. and Goel, A. K. (2001). Learning about constraints by reflection. In Stroulia, E. and Matwin, S., editors, *Proceedings of the Fourteenth Canadian Conference on Artificial Intelligence - AI-01*, pages 131–140, Ottawa, ON, Canada.

[Murdock et al., 1998] Murdock, J. W., Goel, A. K., Donahoo, M. J., and Navathe, S. (1998). Method specific knowledge compilation: Towards practical design support systems. In *Proceedings of the Fifth International Conference on Artificial Intelligence and Design - AID-98*, Lisbon, Portugal.

[Murdock et al., 1997] Murdock, J. W., Shippey, G., and Ram, A. (1997). Case-based planning to learn. In *Proceedings of the Second International Conference on Case-Based Reasoning - ICCBR-97*, Providence, RI.

[NCSA, 1997] NCSA (1997). Mosaic for X wish list. http://www.ncsa.uiuc.edu/SDG/Software/XMosaic/wish-list.html. Accessed September 1998.

[Neumann, 1984] Neumann, O. (1984). Automatic processing: A review of recent findings and a plea for an old theory. In Prinz, W. and Sanders, A. F., editors, *Cognition and the motor processes*, pages 255–293. Springer-Verlag, Berlin.

[Newell and Simon, 1963] Newell, A. and Simon, H. (1963). GPS, a program that simulates human thought. In Feigenbaum, E. A. and Feldman, J., editors, *Computers and Thought*. R. Oldenbourg KG.

[Nilsson, 1998] Nilsson, N. J. (1998). *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, San Francisco, CA. Errata from http://www.mkp.com/nils/clarified. Accessed May 2001.

[Pearl, 1988] Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, San Francisco, CA.

[Penberthy and Weld, 1992] Penberthy, J. S. and Weld, D. (1992). UCPOP: A sound, complete, partial-order planner for ADL. In *Third International Conference on Knowledge Representation and Reasoning - KR-92*, Cambridge, MA.

[Pérez and Carbonell, 1994] Pérez, M. A. and Carbonell, J. G. (1994). Control knowledge to improve plan quality. In *Proceedings of the Second International Conference on AI Planning Systems*, Chicago, IL.

[Quinlan, 1986] Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1:81–106.

[Rugaber and Guzdial, 1999] Rugaber, S. and Guzdial, M. (1999). Ectropic software. In *International Conference on Software Engineering (ICSE-99) Workshop on Software Change and Evolution*, Los Angeles.

[Russell and Wefald, 1991] Russell, S. and Wefald, E. (1991). *Do the right thing: Studies in limited rationality*. Artificial Intelligence. MIT Press, Cambridge, Mass.

[Sacerdoti, 1974] Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135.

[Schreiber et al., 2000] Schreiber, G., Akkermans, H., Anjewierden, A., de Hoog, R., Shadbolt, N., de Velde, W. V., and Wielinga, B. (2000). *Knowledge Engineering and Management: The CommonKADS Methodology*. MIT Press, Cambridge, MA.

[Sembugamoorthy and Chandrasekaran, 1986] Sembugamoorthy, V. and Chandrasekaran, B. (1986). Functional representation of devices and compilation of diagnostic problem solving systems. In Kolodner, J. and Riesbeck, C., editors, *Experience, Memory and Reasoning*, pages 47–73. Erlbaum, Hillsdale, NJ.

[Shippey et al., 1998] Shippey, G. T., Murdock, J. W., and Ram, A. (1998). PLUTO: Managing multistrategy learning through planning. In *Abstract in Proceedings of the Fifteenth National Conference on Artificial Intelligence - AAAI-98*, pages 1201–1201.

[Stefik, 1981] Stefik, M. (1981). Planning and meta-planning (MOLGEN: Part 2). *Artificial Intelligence*, 16(2).

[Stroulia, 1994] Stroulia, E. (1994). *Failure-Driven Learing as Model-Based Self Redesign*. PhD dissertation, Georgia Institute of Technology, College of Computing.

[Stroulia and Goel, 1995] Stroulia, E. and Goel, A. K. (1995). Functional representation and reasoning in reflective systems. *Journal of Applied Intelligence*, 9(1):101–124. Special Issue on Functional Reasoning.

[Stroulia and Goel, 1996] Stroulia, E. and Goel, A. K. (1996). A model-based approach to blame assignment: Revising the reasoning steps of problem solvers. In *Proceedings of the National Conference on Artificial Intelligence - AAAI-96*, Portland, Oregon.

[Stroulia and Goel, 1997] Stroulia, E. and Goel, A. K. (1997). Redesigning a problem-solver's operators to improve solution quality. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence - IJCAI-97*, pages 562–567, San Francisco. Morgan Kaufmann Publishers.

[Stroulia and Sorenson, 1998] Stroulia, E. and Sorenson, P. (1998). Functional modeling meets meta-CASE tools for software evolution. In *Proceedings of the International Workshop Software Program Evolution*.

[Suchman, 1987] Suchman, L. (1987). *Plans and Situated Actions: The Problem of Human Machine Communication*. Cambridge University Press.

[Sutton, 1988] Sutton, R. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44.

[Tate et al., 1990] Tate, A., Hendler, J., and Drummond, M. (1990). A review of AI planning techniques. In Allen, J., Hendler, J., and Tate, A., editors, *Readings in Planning*, pages 26–49. Morgan Kaufmann, San Mateo, California.

[Veloso, 1994] Veloso, M. (1994). PRODIGY / ANALOGY: Analogical reasoning in general problem solving. In *Topics in Case-Based Reasoning*, pages 33–50. Springer Verlag.

[Veloso et al., 1995] Veloso, M., Carbonell, J., Pérez, A., Borrajo, D., Fink, E., and Blythe, J. (1995). Integrating planning and learning: The PRODIGY architecture. *Journal of Theoretical and Experimental Artificial Intelligence*, 7(1).

[Waters, 1999a] Waters, R. (1999a). ACME server user's manual. http://www.cc.gatech.edu/morale/tools/acme/acmeservman.htm. Accessed June 2001.

[Waters, 1999b] Waters, R. (1999b). ACME visual editor (VisEd) user's manual. http://www.cc.gatech.edu/morale/tools/acme/visedman.html. Accessed June 2001.

[Watkins and Dayan, 1992] Watkins, C. J. C. H. and Dayan, P. (1992). Technical note: Q-learning. *Machine Learning*, 8(3).

[Weld et al., 1998] Weld, D. S., Anderson, C. R., and Smith, D. E. (1998). Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence - AAAI-98*, pages 897–904. AAAI Press.

[Wilensky, 1981] Wilensky, R. (1981). Meta-planning: Representing and using knowledge about planning in problem solving and natural language understanding. *Cognitive Science*, 5(3):197–233.