Journal of Artificial Intelligence Research 1 (under submission) ?-? Submitted 12/04; published ??/??

Self-Improvement through Self-Understanding

J. William Murdock MURDOCKJ@US.IBM.COM
IBM Watson Research Center
19 Skyline Dr.
Hawthorne, NY 10532

Ashok K. Goel GOEL@CC.GATECH.EDU
Artificial Intelligence Laboratory
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

Abstract

The ability to adapt is a key characteristic of intelligence. This work investigates model-based reasoning for enabling intelligent software agents to adapt themselves as their functional requirements change incrementally. This article 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 teleology. A TMKL model of an agent explicitly represents the tasks the agent addresses, the methods it applies, and the knowledge it uses. These models are used in a reasoning shell called REM (Reflective Evolutionary Mind). REM enables the execution and incremental adaptation of agents which contain TMKL models of themselves.

1. Introduction

An intelligent agent may encounter situations for which it is important to extend its range of capabilities. 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 a new type. As a third example, in the domain of meeting scheduling, a system may fail to schedule a meeting because the system 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.

Over the life of an agent, changes to the requirements placed on an agent may concern many aspects, e.g., performance, constraints, resources, and functionality. For example, one could demand that an agent perform some task more quickly. Another kind of change involves a modification of the resources available to a planner; for example, an agent built to disassemble physical objects using a certain set of tools such a screwdrivers, hammers, etc., may need to adapt when new tools are made available to it or existing tools are taken away from it. In this article, we focus on one specific kind of requirement change: incremental

changes to the functionality of the agent, specifically the addition of a new task to the agent, e.g., adding the task of assembly to a disassembly agent.[1]

Since the functionality desired of the agent changes only incrementally, it follows that the new task is related to a task that agent already knows about and can accomplish. Note that case-based reasoning addresses a different problem. Given a plan to disassemble one kind of device and the goal of disassembling another, similar device, a case-based planner may adapt the known plan to accomplish the new goal. In fact, this is precisely what ADDAM, a case-based agent for physical device disassembly (Goel, Beisher, & Rosen, 1997), does. In contrast, the present work addresses the problem of adapting the *agent's* design, i.e., the architecture of its reasoning processes for accomplishing its tasks.

Thus, the techniques described in this work can be viewed as constituting *meta-case-based reasoning*. Given a planner for generating disassembly plans for a class of devices and the goal of assembling a specific device in that class, while the desired output in case-based reasoning is a specific plan for the particular assembly goal, the desired output in meta-case-based reasoning is a new reasoning process for generating assembly plans for the device class more generally (in addition to a specific plan for the particular assembly goal). While in case-based reasoning, the planner for disassembling devices may repeatedly adapt a disassembly plan each time a new assembly goal is presented, in meta-case-based reasoning, once the reasoning process of the planner for generating disassembly plans is adapted to generate assembly plans, the planner may directly invoke its method for generating an assembly plan each time a new assembly goal is presented (and invoke its method for generating a disassembly plan when a disassembly goal is presented to it). Note that while the meta-level reasoner may be viewed as case-based, the object-level reasoner, e.g., the planner for generating disassembly plans, need not be case-based.

The class of problems for which case-based reasoning is applicable is such that the solution desired for a new problem in the class is structurally similar to the known solution to a familiar problem. In contrast, the class of problems for which meta-case-based reasoning is applicable is such that reasoning process required for addressing a new problem in the class is structurally similar to the process required to address a familiar problem but the solution to the new problem is not necessarily similar to the solution to the familiar problem.

Adaptation of an agent's reasoning processes for accomplishing a new task raises several issues:

- Which of the agent's existing tasks is similar to the new task?
- What are the differences between the effects of the new and existing tasks?
- What elements of the agent's design contribute to those differences?
- What modifications can be made to the agent to eliminate those differences?

These are all challenging questions. Identifying tasks that accomplish similar effects is challenging because the effect of some task may be abstract and complex; for example, the assembly task has an effect that a device is *assembled*, which is operationalized for a

1. Although we use assembly/disassembly as an illustrative example in this paper, our work addresses a larger range of problems as evidenced by the logistical planning example described later.

particular device in terms of connections among that device's components. Furthermore, there are many dimensions along which abstract effects may be similar, and only some of these dimensions are likely to be useful in identifying tasks that are appropriate for adaptation. Computing the differences between effects of tasks involves similar difficulties in working with abstract and possibly ill-defined specifications. Determining which elements in the agent's design contribute to the differences can be challenging because an agent is likely to have many parts and the contribution that a particular element makes to the ultimate result may be complex. Lastly, determining what modifications to make to an element once it is found can be challenging because a piece of a computation can have a large 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). Identifying the elements to modify and the modifications to make to them is particularly challenging when the following conditions hold: (1) there is no simple correspondence between task differences and the required modification, (2) the modification space is very large, and (3) a modification may have many effects, some of which may percolate throughout the agent's design.

To address the issue of identifying a candidate element for modification, it is helpful to have knowledge about how the various elements of the agent contribute to the overall effect of that agent. The issue of identifying a candidate modification to a candidate element can be addressed by 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. Thus, the challenges posed by adaptation in response to new tasks suggest that an agent's model of itself should represent its composition and teleology, and in particular:

1. What the elements of the agent do.
2. What the intended effect of the agent is.
3. How the elements 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 is implemented by one or more methods, and the methods are composed of lower level tasks. 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).

REM (Reflective Evolutionary Mind) is a shell for running and modifying agents encoded in TMKL. Figure 1 presents the architecture of REM. There are two main components of REM: an execution module and an adaptation module. The execution module allows an

agent encoded in TMKL to run; it works by stepping through the tasks and the methods specified in the model and applying the indicated knowledge. The adaptation module modifies the TMKL model of an agent. Because the execution of the agent is performed by stepping through the model, changes to the model directly alter the behavior of the agent. For example, if a new method for a known task is added to the model during adaptation, then that method is available the next time the execution module is selecting a method for that task.

To activate REM, a user provides two kinds of inputs: a task to be performed and a set of parameter values for that task. For example, the user of a disassembly agent might specify disassembly as the task and a specific device (such as a computer or a camera) as the value for the input parameter for that task. When such a request is made, REM first checks the model of the agent to see if the task specified is one for which the model provides a method. If it is, REM can immediately invoke its execution module to perform the desired task. If not, REM must invoke its adaptation module to modify the model so that it has a method for the given task. For example, if the user of the disassembly agent wants that agent to instead assemble a device, REM first performs adaptation to build a method for assembly (i.e., an implementation for that task). If the adaptation module is able to build a method, the execution module can then perform the desired task. Whenever execution is performed, if it is successful, then the agent is done. If, however, execution is unsuccessful, then further adaptation is needed; a trace generated during execution can help support this adaptation. The cycle of execution and adaptation continues until either execution has succeeded or all applicable adaptation techniques have failed.

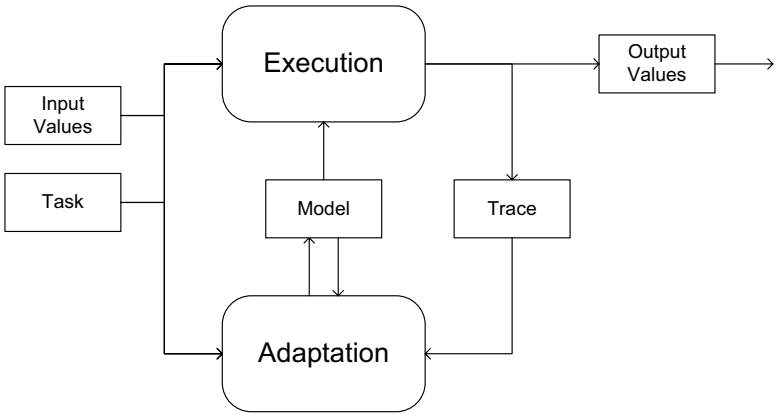

Figure 1: The architecture of REM. Large round boxes indicate procedural elements. Small boxes indicate information used and / or produced by those elements.

As implied by Figure 1, there are two distinct situations for which REM can employ adaptation:

Proactive adaptation occurs before any execution. It is performed when a new task (with no methods) is provided to REM. Its only inputs are the description of the task and the input parameter values provided to that task.

Retrospective adaptation occurs after unsuccessful execution of a task that already has one or more methods. Its inputs include not only the task and the input values but also the model (methods, subtasks, etc.) for the task and a trace of the execution process.

REM has four different strategies for performing adaptation.

Situated learning involves simply trying arbitrary actions and then gradually learning a policy that leads to the desired result (as specified by the given task). REM only uses this strategy for proactive adaptation.

Generative planning involves determining a complete sequence of actions that accomplish the desired result. REM can use this strategy for proactive adaptation. In addition, REM can use this strategy for retrospective adaptation by creating a plan for a single failed subtask.

Fixed-value production involves adding a highly specialized subtask to a model that provides a fixed value for an output parameter in a specific situation. This strategy can only be used for retrospective adaptation because it requires information about a specific situation, encoded in a trace.

Relation mapping involves finding some known task whose intended effect is directly related to the effect of the given task via a single relation and then constructing a new method for the given task by adapting a method for the known task. REM uses this strategy for proactive adaptation because it involves the creation of an entirely new method for a task.

In this paper, we focus primarily on the use of relation mapping in proactive adaptation; the other strategies are also described, but in less detail. The experimental results presented in Section 6.1 primarily illustrate how relation mapping contrasts with situated learning and generative planning. Relation mapping makes extensive use of an existing model for a known task, so it has a fairly significant knowledge requirement. In contrast, situated learning and generative planning require only knowledge of states and actions. In addition, relation mapping requires extensive reasoning about the model; for large models this can impose some overhead that situated learning and generative planning avoid. However, the knowledge requirements for relation mapping, while significant, are frequently obtainable. Furthermore, our experimental results and theoretical analysis show that while the overhead imposed by reasoning about models can make relation mapping slower than the alternatives for simple problems, relation mapping provides enormous speed benefits for more complex problems.

There are three key elements to REM's approach to supporting flexible reasoning and behavior: (1) a language for modeling intelligent systems, (2) an algorithm for executing these systems, and (3) a variety of adaptation strategies capable of modifying these systems. The results presented in this paper provide validation for this approach by specifically showing how one of REM's adaptation strategies can use its modeling language to enable successful execution.

2. Agent Models

Systems in REM are modeled using the Task-Method-Knowledge Language (TMKL). TMKL systems are divided into tasks, methods, and knowledge. A task is a unit of computation which produces a specified result. A task answers the question: *what* does this piece of computation do? A method is a unit of computation which produces a result in a specified manner. A method answers the question: *how* does this piece of computation work? Tasks encode functional information; the production of the specified result is the function of a computation. The knowledge portion of the model describes the different concepts and relations that tasks and methods in the model can use and affect as well as logical axioms and other inferencing knowledge involving those concepts and relations. Formally, a TMKL model consists of a tuple (T, M, K) in which T is a set of tasks, M is a set of methods, and K is a knowledge base. For more details (including examples) of all three portions of a TMKL model, see (Murdock, 2001). A relatively concise overview of this material is provided below.

2.1 Tasks

A task in TMKL is a tuple $(in, ou, gi, ma, [im])$ encoding input, output, given condition, makes condition, and (optionally) an implementation respectively. The input (in) is a list of parameters that must be bound in order to execute the task (e.g., a task involving movement typically has input parameters specifying the starting location and destination). The output (ou) is a list of parameters that are bound to values as a result of the task (e.g., a task that involves counting a group of objects in the environment will typically have an output parameter for the number of objects). The given condition (gi) is a logical expression that must hold in order to execute the tasks (e.g., that a robot is at the starting location). The makes condition (ma) is a logical expression that must hold after the task is complete (e.g., that a robot is at the destination). The optional implementation (im) encodes a representation of how the task is to be accomplished. There are three different types of tasks depending on their implementations:

Non-primitive tasks each have a set of methods as their implementation.

Primitive tasks have implementations that can be immediately executed. TMKL allows three different implementations of this sort: an arbitrary Lisp function, a logical assertion that is entered into the current state when the task is invoked, or a binding of an output parameter to a query into the knowledge base. Some primitive tasks in TMKL are *actions*, i.e., their indicated effects involve relations that are known to be external to the agent. Primitive tasks that are not actions involve internal computation only.

Unimplemented tasks cannot be executed until the model is modified (by providing either a method or a primitive implementation).

Table 1 presents an example TMKL task called execute-remove. This task appears in the ADDAM disassembly planning agent (see Section 3 for more details on ADDAM). Performing this task removes a component from a device. The task has one input, the

Table 1: An example of a task in TMKL

```
(define-task execute-remove
  :input (current-object)
  :given (:and
          (physical-component (value current-object))
          (present (value current-object))
          (free (value current-object)))
  :by-procedure do-remove
  :asserts (:not (present (value current-object))))
```

object to be removed, and no outputs. The given expression indicates that the task can only be performed when its argument is a physical component, and is present in the current device, and is free (i.e., is not fastened to or blocked by any other components). The implementation of this task includes both a procedure (do-remove) and a logical assertion (that the object is no longer present).

The define-task form in TMKL is represented internally as a set of logical assertions, e.g., (task⟩asserts execute-remove '(:not (present (value current-object)))). It is expected that most of the information in a TMKL model will be encoded using these 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 individual assertions.

2.2 Methods

A method in TMKL is a tuple (pr, ad, st) encoding a provided condition, additional results condition, and a state-transition machine. Like the given and makes conditions of a task, these conditions specify assertions that must be true before and after execution. However, while the conditions on a task indicate the function that the task is intended to accomplish, the conditions on a method encode only incidental requirements that are specific to that particular way of accomplishing that function. For example, a method for movement that involved driving a car would have a provided condition that a car be available and have enough gasoline and an additional results condition that the car has moved and has consumed gasoline. The division of effects into functional effects (associated with a task) and incidental effects (associated with a method) is useful in guiding the adaptation of tasks; a new or modified method for a given task is required to satisfy the functional specification of that task (i.e., its given and makes conditions) but may have entirely new incidental requirements.

The state-transition machine in a method contains states and transitions. States each connect to a lower-level task and to a set of outgoing transitions. Transitions each contain an applicability condition, a set of bindings of output parameters in earlier subtasks to input parameters in later subtasks, and a next state that the transition leads to. The execution of a method involves starting at the first transition, going to the state it leads to, executing the subtask for that state, selecting an outgoing transition from that state whose applicability condition holds, and then repeating the process until a terminal transition is reached.

2.3 Knowledge

The representation of knowledge (K) in TMKL is done via Loom (MacGregor, 1999), an off-the-shelf knowledge representation (KR) framework. Loom provides not only all of the KR capabilities found in typical AI planning system (the ability to assert logical atoms, to query whether a logical expression holds in the current state, etc.) but also an enormous variety of more advanced features (logical axioms, truth maintenance, multiple inheritance, etc.). In addition, Loom provides a top-level ontology for reflective knowledge (including terms such as thing, concept, relation, etc.).

TMKL adds a variety of terms to the Loom ontology. Some involve the core elements of a TMKL model: task, method, and various components of tasks and methods (e.g., parameter). In addition, TMKL provides a variety of additional terms for representing domain information that is frequently useful for reflection. For example, there are four TMKL terms that represent traits that a relation can have: external-state-relation, internal-state-relation, external-definition-relation, internal-definition-relation. 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. 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.

Two other terms that TMKL adds to the Loom ontology are similar-to and inverse-of. 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.

Both similar-to and inverse-of are relations that can either be computed by the system or asserted directly by the developer of the model. 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 (i.e., a logical axiom). REM does not have any built-in implies rules for inverse-of but it does have a couple of these rules for similar-to.[2] 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

2. REM's built-in rules for similar-to include the fact that tasks with identical parameter lists and similar given and makes assertions are similar. Similarity for assertions is implied by having identical logical structure and identical or similar instances.

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.

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 5.4 finds a relation that 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.

3. ADDAM: An Illustrative Agent Example

One agent which has been modeled in TMKL and executed in REM is ADDAM (Goel et al., 1997).[3] ADDAM is a physical device disassembly agent. [4] ADDAM was not created as part of this research; the original ADDAM system did not contain any representation of itself and thus did not have any ability to reflect on its own reasoning. Thus ADDAM is a legacy agent for the purposes of the REM project. In our research on REM, we constructed a TMKL model for ADDAM, connected that model to relevant pieces of the existing ADDAM code, and conducted experiments involving the execution and adaptation of ADDAM in REM (e.g., having REM use its model of ADDAM to change the disassembly planning algorithm into an assembly planning algorithm).

There are two primary reasons why disassembly is a difficult task to automate: (1) the entire sequence of physical movements for disassembling a device is extremely long and complex, and (2) 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 actually be useful for automation, a disassembly plan does need to ultimately be translated into robotic movements. ADDAM has a separate module that addresses this issue by translating individual actions from the high level plan into combinations of movements for a detailed robot simulator. The experiments of ADDAM within REM have not involved this module at all because it is severely limited, is not part of the main ADDAM system, and is not particularly relevant to the issues in the REM/ADDAM experiments.

Reasoning about high-level actions does not completely resolve the second issue: the cost of combining actions to form a goal. Some fairly simple devices are prohibitively expensive to disassemble without prior knowledge, even when represented and reasoned about at the level of detail in ADDAM (as demonstrated by experiments described in Section 6.1). The

3. Several other agents have also been run within REM; these other agents are not described in detail in this article, but are outlined briefly at the start of Section 6.
4. ADDAM is an acronym for Assembly/Disassembly Description And Modification. ADDM was initially envisioned as generating both disassembly and assembly plans, and hence its name. The actual desigin and implementation of ADDAM however covered only disassembly; thus, ADDAM by itself cannot generate assembly plans.

approach that ADDAM takes to this problem is case-based reasoning; ADDAM adapts old plans for disassembling devices into new plans for disassembling similar devices.

3.1 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.

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. ADDAM's process for adapting plans 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 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. 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.

The process employed by ADDAM involves first creating a new disassembly plan and then executing that plan. 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 involves constructing a mapping between elements of the new and existing devices, converting portions of the plan for the existing device into analogous portions of the plan for the new device, and then converting the ordering dependencies for the existing plan into analogous ordering dependencies for the new plan.

4. Algorithm for Execution

The algorithm for execution in REM involves recursively stepping through the hierarchy of tasks and methods. Given a non-primitive task, REM selects a method for that task and executes that method. Given a method, REM begins at the starting state for that method, executes the lower level task for that state, and then selects a transition which either leads to another state or concludes the method. At the lowest level of the recursion, when REM encounters a primitive task it simply performs that task.

There are three additional issues regarding the execution algorithm: selection, trace generation, and failure monitoring. The issue of selection comes up when choosing a method for a non-primitive task and when choosing a transition within a method. Both methods and transitions have logical conditions which specify when they are applicable; if more than one option has its logical condition satisfied at any point in the process, then reinforcement learning, specifically Q-learning (Watkins & Dayan, 1992), is used to choose among those options. When the desired effect for the main task has been accomplished, positive reinforcement is provided to the system, allowing it to learn to choose options which tend to lead to success. For large problems, Q-learning requires an enormous amount of experience to develop an effective policy; gathering this experience can be very expensive. Consequently, it is recommended that models in TMKL be designed in such a way that *most* decision points have mutually exclusive logical conditions. If a very small number of decision points are left for Q-learning to address, it can develop a policy for those decision points with relatively little experience.

REM performs trace generation during execution because some of REM's adaptation strategies (see Sections 5.2 and 5.3) can then use the trace to identify failures that occurred during execution. Trace generation is addressed in REM by producing a record of what happened at each step of the process. The steps of the trace are encoded as knowledge structures in Loom, and are linked directly to the TMKL tasks, methods, states, transitions, etc. These traces are used if adaptation is required after execution; the traces provide a particular path through the model and thus can guide and constrain the search for elements in the model that may need to be modified in response to the execution.

The issue of failure monitoring arises in REM because tasks and methods contain explicit information about what they do, making it possible to detect when they are not behaving correctly. For example, a task has a slot, makes, which contains a logical condition that must hold after the task is complete. If this condition does not hold, execution terminates, and an annotation is added to the step in the trace for that task indicating that the makes condition was not met. In addition, annotations are also added to the higher level tasks and methods from which the task was invoked if their indicated results have also not been produced. Failure annotations are useful for signifying that some adaptation needs to occur and also for guiding the search for locations that require adaptation.

5. Algorithms for Adaptation

When an agent is not able to do what the user wants done, adaptation needs to be performed. Adaptation results in a revised model that can then be executed using the mechanisms described in Section 4. There are four general varieties of adaptation which have been performed in this research: situated learning, generative planning, fixed-value production, and relation mapping. 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.

5.1 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 that contains no specific knowledge about how to address the task; the method makes the widest variety of actions possible at any given time. When the method is executed, all decisions about what to do are made using a policy developed by the reinforcement learning mechanism.

The algorithm for situated learning adaptation begins by creating a new method for the main task. Next, a new subtask is created for the method; the method specifies that the subtask is executed repeatedly until the makes condition for the main task is met. The new subtask is then given methods for each possible combination of primitive actions and input values that exist in the environment. For example, there is an action in the disassembly domain called execute-remove which has one input parameter, current-object. If some task were requested with inputs which contained (for example) five objects, then five methods would be created each of which invokes that action on one of the objects. The provided conditions attached to the methods are simply the given conditions for the actions applied to the specific values. For example, if the inputs of an action include an object called board-1 and the given condition for that action is (present (value current-object)) then the provided condition for the method that invokes the action on that object is (present board-1).

The result of this algorithm is a new method for the main task which runs in a continuous loop performing some action and then checking to see if the desired result has been obtained. This method allows any possible action to be performed at each step, so all of the decision making required for the process is deferred until execution time.

5.2 Generative Planning Adaptation

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. REM uses Graphplan[5] (Blum & 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.

5. 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 article 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.

The algorithm for generative planning in REM consists of three parts. In the first part, the task, primitive actions, and input values provided to REM are translated into a format which the planner can process. In the second part, the planner is invoked on that information. In the third part, the resulting plan is translated into a TMKL method for the given task.

The translation from REM to Graphplan in the first part of the process is fairly straightforward. The makes condition of the task is translated into the goal, substituting the actual input values for references to parameters. The given and makes conditions of the primitive actions are translated directly into preconditions and postconditions for Graphplan operators. All facts in the knowledge base which involve relations that are referenced in an operator are translated into Graphplan facts. Note that this translation may be incomplete. Loom's language for representing queries and assertions is much more expressive than Graphplan's language for representing conditions and facts. Loom supports constructs such as subconcepts and inheritance, universal and existential quantification, etc. If REM is unable to translate some of its facts or primitive actions into a format which Graphplan can use, it omits them and runs Graphplan with the information that it can translate. This can lead to an incorrect plan or no plan at all (which can, in turn, require further adaptation using one of REM's other mechanisms).

The translation of the plan returned by Graphplan into a TMKL method for REM is somewhat more subtle that the translation from REM to Graphplan. Some parts of the translation are very direct: the plan directly specifies the actions and their order. The more complex part of the translation involves inferring the links between the parameters. If a step in the plan involves a value which has not been used in an earlier action and is an input parameter to the main task, then the input parameter to the main task is linked to the corresponding parameter in the primitive action. Additional references to the same value involve links from the parameter in the earlier action to the parameter in the later action. Thus the method produced can be reused in later situations involving the same task with different values for the parameters.

The process of extracting a TMKL method from a Graphplan plan is inspired by Explanation-Based Generalization (Mitchell, Keller, & Kedar-Cabelli, 1986). Specifically, it generalizes the plan into a method by inferring relationships among task parameters; the relationships are encoded as links in the method's transitions. For example, if a specific component in a plan is put into place and then attached to that place, REM will infer a binding between the parameter in the placement action and the corresponding parameter in the attachment action. The resulting method is 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 using specific values, accomplish the overall goal of the main task, which is typically expressed in terms of the task's parameters.

In future work, we plan to conduct experiments with REM exporting actions and states to PDDL (McDermott, 1998), a standardized language for specifying planning problems. This will allow us to use any of a wide variety of recent planning systems in place of Graphplan. Because PDDL is much more expressive than Graphplan's operator language, such a mechanism could allow a wider variety of actions to be exported. The primary prerequisite

that a planning system must have to be used in this mechanism is minimal requirements for control knowledge. REM invokes the generative planning adaptation mechanism when it has missing or inapplicable task decomposition knowledge; thus planners that require such knowledge (e.g., Hierarchical Task Network planners) are not appropriate for this mechanism.

5.3 Fixed-Value Production

The fixed-value production strategy is a model-based mechanism that can only be used for retrospective adaptation. Its inputs are a model, a failed trace of that model, and explicit feedback from the user about a knowledge item that should have been considered during the execution recorded in the trace.

The algorithm for fixed-value production in REM begins by computing the differences between the actual results produced during execution and desired results (as indicated by the feedback). Next it searches the trace for events which directly relate to the computed differences. For example, if the actual result involved having board-1 removed from a device, and the desired result involved having board-2 removed from the device, then the search process would identify portions of the process which involve selecting a component to remove. Lastly, the fixed-value production mechanism adds a new subtask to the model; this subtask causes the identified portion of the process to produce the desired result whenever it is executed under the same circumstances (e.g., adding a new primitive task which produces the value board-2 whenever the same task is performed on the same device).

5.4 Relation Mapping

If REM has a model, but that model does not include a method for the task requested, then it must perform adaptation proactively (i.e., before attempting execution). The relation mapping strategy is a model-based mechanism for proactive adaptation. As an example, if REM is given a description of assembly and some assembly actions and a model of a process for disassembly, it can try to transfer its knowledge of disassembly to the problem of assembly. Figure 2 shows some of the tasks and methods of the ADDAM hierarchical case-based disassembly agent which are particularly relevant to this example; the complete ADDAM model contains 37 tasks and 17 methods. For a more detailed description of that model, see (Murdock, 2001).

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 new 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,

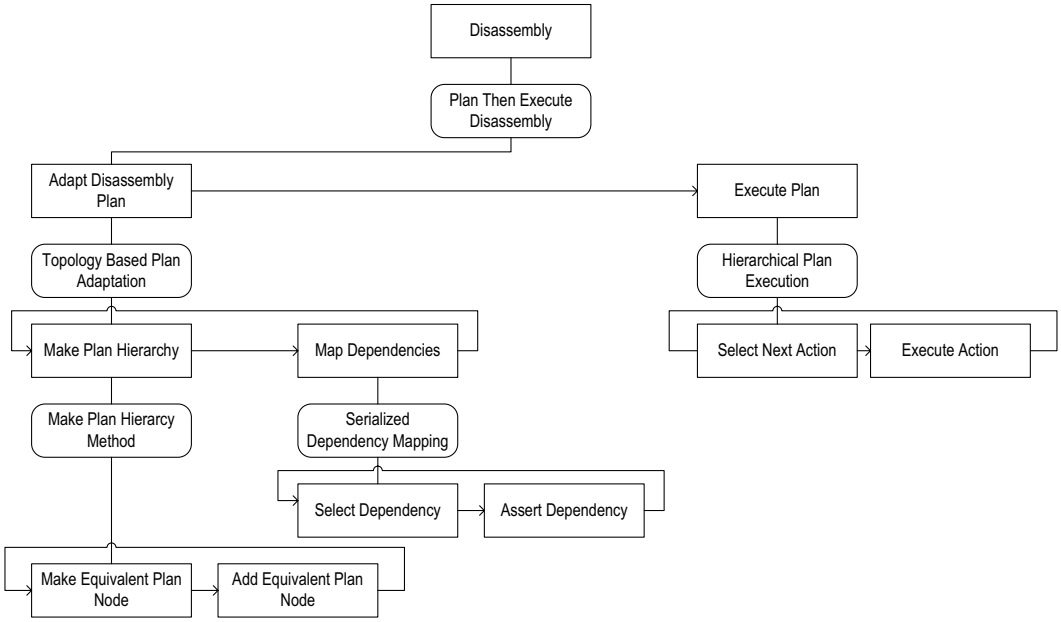

Figure 2: A diagram of some of the tasks and methods of ADDAM.

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 that adds a plan node to a plan is implemented by a 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 the ADDAM disassembly process (encoded as a TMKL model) and is asked instead to assemble a device, it needs to perform some adaptation in order to have a method for assembly. In this situation, any of three proactive techniques (situated learning, generative planning, and relation mapping) can 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.

Table 2 presents the algorithm for relation mapping. The first step in the process involves retrieving an existing task that is connected to the given task via the similar-to relation. 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 that do not match are directly linked via a single relation.

Table 2: Algorithm for relation mapping

Algorithm **relation-mapping**(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
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:implementation = COPY known-task:implementation
map-relation = [relation that 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
```

Once a related task has been retrieved, the relation mapping algorithm copies 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. The algorithm requires that the selected relation be a binary relation defined in either in the specific domain of the agent or in the general TMKL ontology; from this set of candidates one is chosen that maps values from the makes condition of the main task to the makes condition of the known task. In the assembly example, the relation which is used is inverse-of; other experiments have employed the relation-mapping strategy with other relations, such as generalization and specialization (for example, see Section 6.2). For the assembly problem, the inverse-of 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.

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 for inverse-of. 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. Specifically, the state transition machine for the method is altered to add one new state, which executes the inserted task, and three new transitions. One of the transitions leads into the inserted task, one leads out of it, and one goes *around* it (i.e., directly from the previous state to the next state). When the main-task is later executed, after the subtask in the previous state is done, REM may select either the transition leading to the inserted task or the transition going around the inserted task; this decision is resolved through trial and error (via Q-learning).

Figure 3 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 (e.g., Make Equivalent Plan Node in Table 3) are implemented by procedures, REM is not able to directly modify them. Instead, it inserts a new mapping task in between the primitive tasks (as shown in Table 4). A new state and two new transitions are also added to the corresponding method. 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 that asserts a dependency is modified. Because the primitive task for asserting dependencies is implemented as a simple

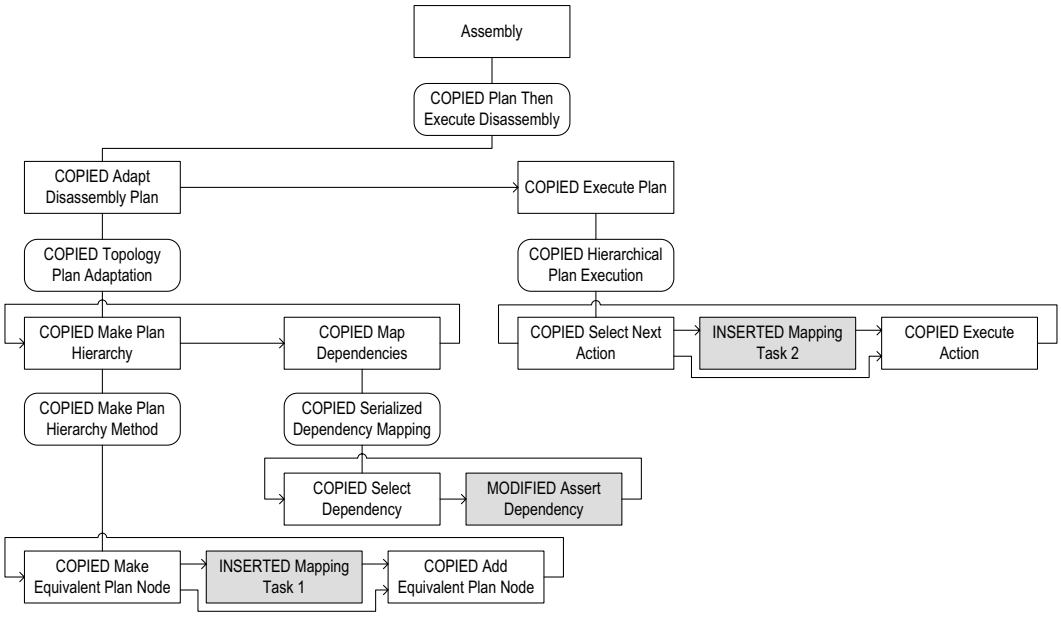

Figure 3: Tasks and methods produced for the assembly process by adapting ADDAM. Tasks which have been added or modified are highlighted in grey.

logical assertion (node-precedes), it is possible to impose the inverse-of relation on that assertion. In the modified model, that relation is replaced by the node-follows relation. Tables 5 and 6 show this task before and after modification. The modification is such that 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.

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 a reflective process to analytically 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

Table 3: A relevant task in the original model that is implemented by a procedure (since REM cannot directly modify the procedure, it does not modify this task but instead inserts an optional mapping task immediately after; see Table 4)

```
(define-task Make-Equivalent-Plan-Node
  :input (base-plan-node parent-plan-node
          equivalent-topology-node)
  :output (equivalent-plan-node)
  :makes (:and (plan-node-parent (value equivalent-plan-node)
                                 (value parent-plan-node))
               (plan-node-object (value equivalent-plan-node)
                                 (value equivalent-topology-node))
               (:implies
                (primitive-action (value base-plan-node))
                (type-of-action
                 (value equivalent-plan-node)
                 (type-of-action
                  (value base-plan-node)))))
  :by-procedure make-equivalent-plan-node-proc)
```

Table 4: The new task that REM inserts immediately after Make Equivalent Plan Node

```
(define-task INSERTED-Mapping-Task-1
  :input (equivalent-plan-node)
  :asserts (type-of-action
            (value equivalent-plan-node)
            (inverse-of
             (type-of-action
              (value equivalent-plan-node)))))
```

Table 5: A relevant task in the original model that is implemented by a logical assertion. REM can directly modify this task (see Table 6).

```
(define-task Assert-Dependency
  :input (target-before-node target-after-node)
  :asserts (node-precedes (value target-before-node)
                          (value target-after-node)))
```

Table 6: Modified task in which node-precedes has been replaced by node-follows

```
(define-task MODIFIED-Assert-Dependency
  :input (target-before-node target-after-node)
  :asserts (node-follows (value target-before-node)
                         (value target-after-node)))
```

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 one of the inserted mapping tasks but not both.

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 6.1.

6. Evaluation

Evaluation of REM has involved a variety of experiments and theoretical analysis. Much of this work has involved the disassembly and assembly problems, using ADDAM. We describe these experiments in detail in the following subsection. In the next subsection, other experiments are described in less detail; citations are provided for more information about those other experiments. The final subsection provides an overview of the theoretical complexity of the various algorithms in REM.

6.1 ADDAM experiments

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. The nested roof example discussed in Section 3 is one which has undergone particularly extensive experimentation in the course of this research. A useful feature (for the purpose of our experimentation) of the roof design is the variability in its number of components. This variability allows us to see how different reasoning techniques compare on the same problem at different scales.

REM's generative planning and situated learning adaptation strategies involve the use of only the *primitive* knowledge items and tasks. For example, when using REM with ADDAM, the information available to REM in generative planning and situated learning 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. When using the relation mapping strategy, REM needs access to the complete TMKL model of the reasoning process. In relation mapping, REM with ADDAM adapts the design of the existing disassembly agent to address the assembly task. Thus the results of running these three adaptation strategies provides an experimental contrast between the use of the model and operating without a model.

The performance of REM on roof assembly problem is presented in Figure 4. The first series involves the performance of REM when using ADDAM via relation mapping. 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.[6] 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 affected by the complexity of the roof design, but to a much smaller extent than generative planning or reinforcement learning techniques are.

In a related experiment, the same reasoning techniques are used 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. Figure 5 shows the relative

6. The starting states of the problems ranged from 3 to 39 logical atoms. There is one flaw in the Graphplan data, as noted on 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.

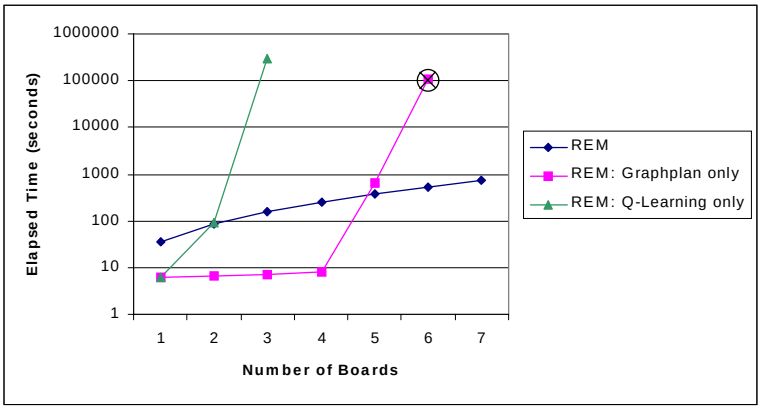

Figure 4: Logarithmic scale graph of the relative performances of different techniques within REM on the roof assembly example (which involves many goal conflicts) for a varying number of boards. The "X" through the last point on the Graphplan line indicates abnormal termination (see text).

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.

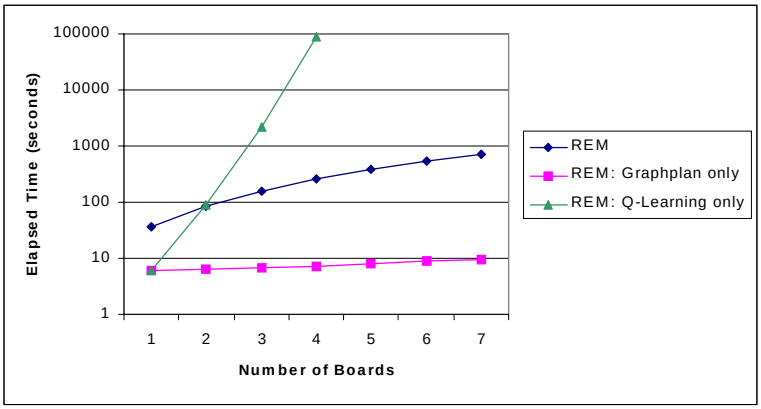

Figure 5: Logarithmic scale graph of the relative performances of different techniques within REM on a modified roof assembly example that has no conflicting goals.

These and other experiments have shown that the combination of model-based adaptation and reinforcement learning on ADDAM provides tractable performance for a variety

of problems that cannot be handled effectively by the alternative approaches. Assembly using pure Q-learning 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, similar devices which involve more complex relationships among the components for which Graphplan is overwhelmed. The adapted ADDAM system is able to handle devices of this sort much more quickly than Graphplan.

6.2 Logistics Experiments

We have conducted experiments with REM in a domain inspired by the Depot domain in the Third International Planning Competition ((2002)); our domain does, however, have significant differences from that domain. The competition domain combines elements of block stacking and route planning (boxes are loaded into trucks, shipped to destinations, and then unloaded at those destinations). Our version uses a much more structured version of the loading problem in which crates each have a different size, can only be stacked in order of size, can only be placed in one of three locations (the warehouse, a loading pallet, and the truck); that part of the problem is isomorphic to the Tower of Hanoi problem, a traditional illustrative example for AI and planning. In addition, we have eliminated the route planning and unloading aspects of the problem. In our version, there is only one possible place to drive the truck, and once the truck arrives, the driver only needs to provide documentation for the shipment (e.g., a purchase order or manifest) rather than unloading. Thus for a planning problem in which two crates start in the warehouse, a complete plan would be: (1) move the small crate from the warehouse to the pallet, (2) move the large crate from the warehouse to the truck, (3) move the small crate from the pallet to the truck, (4) drive to the destination, and (5) give the documentation to the recipient.

Our TMK model for route planning involves four main-subtasks: selecting documentation for the shipment, loading the boxes, driving to the destination, and delivering the object. The only one of those four subtasks that is non-primitive is loading the boxes. That task has two methods which we have labeled Trivial Strategy and Nilsson's Strategy. The former has a provided condition which must hold to be executed: that there be exactly one crate. That strategy simply moves the crate to the truck. Nilsson's Strategy, on the other hand, has no provided condition, i.e., it can be used for any number of crates. It uses a relatively efficient iterative technique created for the Tower of Hanoi problem; specifically, it is a generalization of the Tower of Hanoi strategy in (Nilsson, 1998, errata, p. 4). These two methods are redundant; the behavior of Nilsson's Strategy when there is only one crate is identical to the behavior of Trivial Strategy. However, both methods are included to allow experiments that involve one or the other. For example, we have conducted an experiment using a variation of the example system in which Nilsson's Strategy is removed. This system is able to behave correctly when only one crate is present, but initially fails when more than one crate is present. That failure leads to retrospective adaptation using generative planning, which eventually leads to a correct solution.

The experiment involving the ablation of Nilsson's Strategy involved contrasting the following conditions: (i) REM with a complete model required no adaptation, (ii) REM with the ablated model used generative planning to address the problem of loading boxes

onto the truck and used the existing model to solve the rest of the problem, and (iii) REM with no model used generative planning to solve the entire problem. The results from this experiment (Murdock & Goel, 2003) are consistent with the results in the ADDAM assembly problems: that for harder problems, adaptation of existing hierarchies of tasks and methods provides a substantial advantage over generative (but for easier problems, the overhead involved in adaptation outweighs the benefits). For example, with four boxes, we observed the following performance:

- REM with the complete model took approximately ten seconds.
- REM with the ablated model took approximately an hour.
- REM with no model took approximately four and a half hours hours.

The very large difference between the first two conditions is fairly unsurprising: the loading problem is computationally complex, and Nilsson's specialized iterative strategy solves it much faster than a general-purpose planning algorithm. However, the difference between the last two conditions is much more interesting. The ablated model included just three primitive tasks and indicated that loading boxes occured between the first and second of these tasks. The ablated model had no information on how to load boxes, which was the computationally complex part of the main task. However, by using the model to handle the three simple primitive tasks, REM kept the generative planning system from having to consider how these three primitive tasks interacted with the numerous loading actions. This result illustrates how model-based adaptation can provide substantial benefits by localizing reasoning to a specific subproblem, *even when the subproblem contains nearly all of the complexity.*

We have also conducted experiments in which slightly different tasks were given to REM: delivering boxes with a more general or more specific type of delivery document than the type employed in the existing task. REM has no method for addressing this task, but can adapt its existing method by modifying the first step in the process (selecting a delivery document) to provide more specific or general document. This change is made using REM's relationship mapping strategy over the specialization and generalization relations respectively. This is a very small change to the model, but it illustrates REM's ability leverage a powerful existing model even on a new task that violates a requirement of that model. The model was only built to work with a specific class of delivery documents and fails for more general or more specific documents. If REM had no ability to reflectively adapt, then it would be forced to solve the entire problem using generative planning (or just fail to return a solution). However, the process of loading boxes using Nilsson's Strategy is independent of the particular type of documentation that will be provided once the boxes are delivered, it should be possible to employ the power of this algorithm for this subproblem even when the main problem to be solved is slightly different. REM's relationship mapping strategy enables this behavior.

6.3 Other Experiments

We have also conducted experiments using REM and its predecessors in a variety of other domains. One noteworthy experiment involves adaptation of a mock-up web browsing

system intended to mimic the functionality and behavior of Mosaic 2.4 (Murdock & Goel, 1999). In this experiment, the browser encounters a document of a type that it does not know how to display. A combination of user feedback and retrospective adaptation (using model and trace information) enables the browser to respond to this demand. This experiment is very dissimilar to the assembly/disassembly experiment

Another domain that we have studied is meeting scheduling. We have conducted an experiment in which unexpected demands violate constraints built into an automated meeting scheduling system (Murdock & Goel, 2001). The general approach to addressing this problem is very similar to the approach for the web browsing experiment. The same localization mechanism is applicable to each of these problems, but different modification strategies are required because the components needing modification are different (in particular, the relevant portions of the meeting scheduler involve numerical computations, while the relevant portions of the web browser involve simple logic).

6.4 Computational Complexity

We have performed complexity analyses of the algorithms in REM. Below we present the conclusions which were drawn regarding the running time of the various components of REM under a variety of fairly simple assumptions. Complete details of these analyses (including presentation and justification of all of the assumptions) are beyond the scope of this article; instead see (Murdock, 2001).

- The cost of execution (both before and after adaptation) is potentially unlimited because an arbitrary agent may be arbitrarily expensive to run. It is possible, however, to analyze the extra cost imposed by the execution algorithm on top of the total costs of executing the pieces of the process separately. Given a model of size m producing a trace of size t, worst-case extra cost is $O(t)$ under typical assumptions regarding the structure of the model. An example of one of these assumptions is that the maximum number of parameters for any single task is bounded by a constant. Execution can be $O(m \cdot t)$ for models which violate the typical assumptions.

- 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 a typical external planner.

- Given a trace of size t, the worst-case cost of fixed-value production is $O(t)$ under typical assumptions but may also be affected by the size of the model, the amount of feedback provided, and the number of failures which were detected.

- Given a model of size m, the worst-case cost of relation mapping is $O(m^2)$ under typical assumptions but may be $O(m^3)$ for models of unusual configurations.

The combination of these analytical results and the experimental results presented in Section 6.1 forms the evaluation for the work presented in this article. This evaluation verifies that the representations and reasoning processes presented in earlier sections are able to effectively address a variety of problems including some problems of reasonably large scale.

7. Related Research

7.1 Generative Planning

Generative planning (Tate, Hendler, & Drummond, 1990) involves constructing a sequence of actions which lead from a start state to a goal state. Recent advances in generative planning (e.g., Blum & Furst, 1997; Hoffmann & Nebel, 2001), 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 that they have addressed in the past. Thus the enormous cost of constructing a plan must be incurred over and over.

Generative planning and pure model-based adaptation 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 Section 6.1). 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 flat sequences. However, it is possible to build hierarchical plans in which high-level actions are decomposed into more detailed low-level actions. Hierarchies of actions provide explicit 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 of the efficiency benefits of hierarchical planning (because there is some cost to learning hierarchies) in exchange for weaker knowledge requirements.

Whether plan 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.

One approach to hierarchical planning that is particularly closely related to this research is Hierarchical Task Network (HTN) planning. Models in TMKL resemble HTN's in that both involve tasks that are decomposed by methods into partially ordered sets of subtasks. Of course, REM uses TMKL for modeling and executing reasoning processes while HTN's typically are used for planning. There are two other distinctions between the TMKL and HTN formalisms (e.g., Erol, Hendler, & Nau, 1994):

1) TMKL models may include primitive tasks that are external actions that must be performed immediately in the real world. While such tasks include conditions that specify their effects, those specifications can be incomplete and some effects may not be deterministic. This capability does not require additional syntax and is only a subtle difference in the semantics of the formalism. However, it has significant impact on the methodological conventions that are appropriate for those formalisms. A typical HTN has numerous points at which alternative decisions can be made (e.g., multiple applicable methods for a task or multiple orderings of subtasks for a methods); HTN's tend to be relatively descriptive, i.e., they encode all the different ways that a task may be accomplished rather than prescribing a particular strategy for addressing a task. In contrast, TMKL models tend to be much more prescriptive; most branches or loops in a typical TMKL process have precise, mutually exclusive conditions describing what path to take in each possible state. TMKL does allow some decision points to be unspecified; these decisions are resolved by reinforcement learning. However, since a TMKL system cannot backtrack, and instead must start its reasoning over from the beginning whenever it reaches a failed state, acceptable performance is only possible when there are very few unspecified decision points.
2) TMKL encodes information in tasks and methods that is not directly needed for performing those tasks and methods but that is important for reflective reasoning (e.g., similarity inference rules, makes conditions for top level tasks). Such reasoning includes determining when a system has failed, modifying a system to correct a failure, and using portions of a system to address a previously unknown task. For example, if a new unimplemented task is provided that has similar given and makes conditions to some existing implemented task, REM can reuse parts of the implementation of the existing task to construct an implementation for the new task. In contrast, HTN's are primarily used in systems that assume a complete and correct HTN exists and that only attempt tasks encoded in the HTN. When automatically learning of HTN's is done (Ilghami, Nau, Muñoz-Avila, & Aha, 2002), existing HTN's are not reused and extensive supervision is required.

7.2 Case-Based Planning

Case-based planning involves plan reuse by retrieving and adapting past plans to address new goals (Hammond, 1989). As described above, ADDAM is a case-based planner. In contrast, as mentioned in the introduction, REM's model-based adaptation techniques can be viewed as constituting meta-case-based reasoning.

Case-based planning can be more efficient than generative planning in domains in which there is a plan in memory that is extremely similar to the desired plan and the retrieval method can make use of domain-specific plan indexing (Nebel & Koehler, 1995). If a structurally similar plan is available then it can be tweaked for the current goal. In domains in which causal models are readily available, model-based techniques can be used to adapt the old plan. KRITIK, a case-based system for designing physical devices, for example, uses a structure-behavior-function (SBF) model of a known physical device to adapt its design to achieve device functionalities (Goel et al., 1997).

If a structurally similar plan is not available, case-based planning might still work if plan retrieval and adaptation are partially interleaved and adaptation knowledge is used to retrieve an adaptable plan (Smyth & Keane, 1998). This strategy will not work in the

domain of device assembly because, firstly, ADDAM's disassembly plans for a given device are structurally similar to one another, and, secondly, the causal structure of an assembly plan for any device is different from the causal structure of the disassembly plan for that device. Alternatively, it is possible to combine case-based and generative planning to solve different portions of a planning problem (Melis & 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.

In some domains, the plans available in memory and the desired plan have little structural similarity but the reasoning processes for producing the two kinds of plans are very similar. Consider, for example, the disassembly to assembly adaptation example in Section 5.4. The assembly plans in this example have little structural similarity with the original disassembly plans upon which they were based: there is typically no overlap at all in the operators used because all of the operators are inverted, and while the objects of those operators are similar, they are manipulated in the reverse order. Thus, a disassembly plan cannot be easily adapted into an assembly plan by simple tweaks (which explains why our experiments do not contain a comparison with case-based planning). However, the reasoning 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). When the reasoning processes for the original task and the new task are structurally similar, even if the plans they produce are dissimilar, and a teleological model of the original reasoning process is available, then meta-case-based reasoning can be used to adapt the original reasoning process for the new task. In fact, the model-based techniques for adapting reasoning processes described above are applicable for precisely this class of domains.

In some domains, it is possible to use learning to enhance the original 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 & 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 to have even a single example with which to try to improve the process.

Some case-based reasoning approaches explicitly reason about process. For example, case-based adaptation (Leake, Kinley, & Wilson, 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.

7.3 Machine Learning

Reinforcement learning (Kaelbling, Littman, & Moore, 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. 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; reinforcement learning is very flexible but, unlike model-based adaptation, does not make any use of any deliberation or existing reasoning strategies.

Explanation-Based Generalization (EBG) (Mitchell et al., 1986) is a learning technique that is similar to model-based adaptation in some respects. 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. EBG has been used for learning of robot assembly plans (Segre, 1988). More recently, DerSNLP (Ihrig & Kambhampati, 1997) has used EBG for repairing plans and learning planning knowledge. As described in Section 5.2, REM too uses a variation of EBG within its generative planning mechanism, specifically in generalizing a plan into a reusable TMKL method. However, 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 (or planning) rules.

7.4 Meta-Reasoning

A wide range of past work has looked at the issue of reasoning about reasoning. One research topic in this area 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.

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. MOLGEN's reasoning 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 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 need to 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.

The reasoning architecture used in Guardian and a variety of related agents (Hayes-Roth, 1995) also 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 processing resources. MRS (Genesereth, 1983) is a logical reasoning architecture that uses meta-rules to guide the ordering of inference steps. Anytime algorithms (Boddy & 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 & Wefald, 1991, Chapter 3) computes the expected benefits of applying algorithms versus simply acting. The main difference between model-based adaptation and reflective 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. 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 reflection.

7.5 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 article and existing work on agent modeling is that this work uses agent models for automated self-adaptation to perform previously unknown tasks. Most other agent modeling projects focus on semi-automated processes. For example, CommonKADS (Schreiber et al., 2000) and DESIRE (Brazier, Dunin Keplicz, Jennings, & Treur, 1997) are methodologies for building knowledge systems which use models of agents throughout the development process.

TMKL is particularly closely related to earlier TMK formalisms (Goel & Murdock, 1996; Griffith & Murdock, 1998; Murdock & Goel, 2001) and Generic Tasks (Chandrasekaran, 1986). A particularly significant ancestor of TMKL is the SBF-TMK language in Autognostic (Stroulia & Goel, 1995). Autognostic uses SBF-TMK models to adapt agents that fail to address the task for which they were designed. One of the agents that Autognostic adapts is Router, a case-based navigation planner. Navigation cases in Router are organized around a domain model. Autognostic-on-Router can modify the domain model and thus reorganize the cases. ROBBIE (Fox & Leake, 1995) uses an intentional model for more directly refining case indexing. In contrast, CASTLE (Freed, Krulwich, Birnbaum, & Collins, 1992) uses an intentional model of a chess-playing agent to learn domain concepts such as fork and pin.

Adaptation in REM is used to add new capabilities to an agent. This distinction 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). This difference in perspective significant 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.

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 all or part of a new task.

Autognostic and REM use different formalizations of TMK. REM's TMKL provides much more expressive representation of concepts, relations, and assertions than the SBF-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 5.4 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 SBF-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 queries. 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. 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. Conclusions

Design of a practical intelligent agent involves a trade-off among several factors such as generality, efficiency, optimality, adaptability and knowledge requirements. Much of AI research investigates different points in the multi-dimensional space formed by these factors. Our work focuses on the trade-off between degree of adaptability, computational efficiency and knowledge requirements. We have described an agent design that represents a new point in this three-dimensional space and a novel compromise among adaptability, computational cost, and knowledge conditions. In particular, we have described a model-based technique for agent self-adaptation when the reasoning process required for addressing a new problem is structurally similar to the process required to address a familiar problem but the solution to the new problem is not necessarily similar to the solution to the familiar problem.

The model-based technique for self-adaptation described above is computationally efficient, makes reasonable knowledge requirements, and promises a significant degree of adaptability. Under typically valid assumptions, the worst-case cost of relation mapping is quadratic in the size of the model (i.e., the number of elements in the agent design), and the worst-case cost of fixed-value production is linear in the size of the processing trace.

The knowledge requirements of the model-based technique are also reasonable: since the agents are artificial, it is reasonable to assert that functional and compositional design models generally are available to the designers of the agents. The model-based technique requires only that the agents have these models and provides the TMKL modeling language for encoding them. However, at present the model-based technique is limited to adaptations when only the functional requirements of the agent change incrementally; for example, the relation mapping mechanism requires that the description of a new task be connected to the description of an existing task via a single known relation (e.g., inversion, generalization, specialization).

For simple adaptation problems, the model-based technique successfully identifies both the design elements that need to be changed and the needed changes to them. For complex problems, however, the technique can only form partial solutions; for other parts of a solution, it simply prunes the search space by identifying portions of the agent design that need to be changed and the kinds of changes that are needed. For these problems, the technique can invoke general-purpose methods such as generative planning and reinforcement learning to form complete solutions. Computational experiments with the technique show that for large and complex problems (e.g., planning problems with numerous conflicting goals), the cost of the model-based technique plus localized generative planning and/or reinforcement learning is substantially less than that of either generative planning or reinforcement learning in isolation.

References

Abowd, G., Goel, A. K., Jerding, D. F., McCracken, M., Moore, M., Murdock, J. W., et al. (1997). MORALE – Mission oriented architectural legacy evolution. In *Proceedings international conference on software maintenance 97.* Bari, Italy.

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

Boddy, M., & Dean, T. (1989, August). Solving time-dependent planning problems. In N. S. Sridharan (Ed.), *Proceedings of the eleventh international joint conference on artificial intelligence - IJCAI-89* (pp. 979–984). Detroit, MI, USA: Morgan Kaufmann.

Brazier, F., Dunin Keplicz, B., Jennings, N., & 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)

Chandrasekaran, B. (1986). Generic tasks in knowledge-based reasoning: High-level building blocks for expert systems design. *IEEE Expert*, *1*(3), 23–30.

Erol, K., Hendler, J., & Nau, D. (1994, August). HTN planning: Complexity and expressivity. In *Proceedings of the twelfth national conference on artificial intelligence - AAAI-94.* Seattle, WA: AAAI Press.

Fox, S., & Leake, D. B. (1995). Using introspective reasoning to refine indexing. In *Proceedings of the international joint conference on artificial intelligence - IJCAI-95* (pp. 391–399). Montreal, Canada.

Freed, M., Krulwich, B., Birnbaum, L., & Collins, G. (1992). Reasoning about performance intentions. In *Proceedings of fourteenth annual conference of the cognitive science society* (pp. 7–12). Hillsdale, NJ: Lawrence Erlbaum Associates.

Genesereth, M. R. (1983). An overview of meta level architectures. In *Proceedings of the third national conference on artificial intelligence - AAAI-83* (pp. 119–124). Washington, D.C.

Goel, A. K., Beisher, E., & Rosen, D. (1997). *Adaptive process planning.* (Poster Session of the Tenth International Symposium on Methodologies for Intelligent Systems)

Goel, A. K., & Murdock, J. W. (1996, November). Meta-cases: Explaining case-based reasoning. In I. Smith & B. Faltings (Eds.), *Proceedings of the third european workshop on case-based reasoning - EWCBR-96.* Lausanne, Switzerland: Springer.

Griffith, T., & Murdock, J. W. (1998). The role of reflection in scientific exploration. In *Proceedings of the twentieth annual conference of the cognitive science society.* Madison, WI.

Hammond, K. J. (1989). *Case-based planning: Viewing planning as a memory task.* Academic Press.

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

Hoffmann, J., & Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, *14*, 253–302.

Ihrig, L. H., & Kambhampati, S. (1997). Storing and indexing plan derivations through explanation-based analysis of retrieval failures. *Journal of Artificial Intelligence Research*, 161–198.

Ilghami, O., Nau, D., Muñoz-Avila, H., & Aha, D. W. (2002). CaMeL: Learning methods for HTN planning. In *Proceedings of the sixth international conference on ai planning and scheduling - AIPS-02.* Toulouse, France: AAAI Press.

International Planning Competition. (2002). Competition domains. http://www.dur.ac.uk/d.p.long/IPC/domains.html.

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

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

Leake, D. B., Kinley, A., & Wilson, D. (1995). Learning to improve case adaptation by intropsective reasoning and CBR. In *Proceedings of the first international conference on case-based reasoning - ICCBR-95.* Sesimbra, Portugal.

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

McDermott, D. (1998). *PDDL, the planning domain definition language* (Tech. Rep.). Yale Center for Computational Vision and Control.

Melis, E., & Ullrich, C. (1999). Flexibly interleaving processes. In *Proceedings of the third international conference on case-based reasoning - ICCBR-99* (pp. 263–275). Munich, Germany.

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

Murdock, J. W. (2001). *Self-improvement through self-understanding: Model-based reflection for agent adaptation.* Ph.D. thesis, Georgia Institute of Technology, College of Computing, Atlanta, GA. (http://thesis.murdocks.org)

Murdock, J. W., & Goel, A. K. (1999). Towards adaptive web agents. In *Proceedings of the fourteenth ieee international conference on automated software engineering - ASE-99.* Cocoa Beach, FL.

Murdock, J. W., & Goel, A. K. (2001). Learning about constraints by reflection. In E. Stroulia & S. Matwin (Eds.), *Proceedings of the fourteenth canadian conference on artificial intelligence - AI-01* (pp. 131–140). Ottawa, ON, Canada.

Murdock, J. W., & Goel, A. K. (2003). Localizing planning with functional process models. In *Proceedings of the thirteenth international conference on automated planning and scheduling - ICAPS-03.* Trento, Italy. (To appear.)

Nebel, B., & Koehler, J. (1995). Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial Intelligence*, *76*(1–2), 427–454.

Nilsson, N. J. (1998). *Artificial intelligence: A new synthesis.* San Francisco, CA: Morgan Kaufmann. (Errata from http://www.mkp.com/nils/clarified. Accessed May 2001.)

Pérez, M. A., & Carbonell, J. G. (1994, June). Control knowledge to improve plan quality. In *Proceedings of the second international conference on ai planning systems.* Chicago, IL.

Russell, S., & Wefald, E. (1991). *Do the right thing: Studies in limited rationality.* Cambridge, MA: MIT Press.

Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, *5*(2), 115–135.

Schreiber, G., Akkermans, H., Anjewierden, A., Hoog, R. de, Shadbolt, N., Velde, W. V. de, et al. (2000). *Knowledge engineering and management: The commonkads methodology.* Cambridge, MA: MIT Press.

Segre, A. M. (1988). *Machine learning of robot assembly plans.* Kluwer.

Smyth, B., & Keane, M. T. (1998). Adaptation-guided retrieval: Questioning the similarity assumption in reasoning. *Artificial Intelligence*, *102*(2), 249–293.

Stefik, M. (1981). Planning and meta-planning (MOLGEN: Part 2). *Artificial Intelligence*, *16*(2).

Stroulia, E., & Goel, A. K. (1995). Functional representation and reasoning in reflective systems. *Journal of Applied Intelligence*, *9*(1), 101–124. (Special Issue on Functional Reasoning)

Tate, A., Hendler, J., & Drummond, M. (1990). A review of AI planning techniques. In J. Allen, J. Hendler, & A. Tate (Eds.), *Readings in planning* (pp. 26–49). San Mateo, California: Morgan Kaufmann.

Veloso, M. (1994). PRODIGY / ANALOGY: Analogical reasoning in general problem solving. In *Topics in case-based reasoning* (pp. 33–50). Springer Verlag.

Veloso, M., Carbonell, J., Pérez, A., Borrajo, D., Fink, E., & Blythe, J. (1995). Integrating planning and learning: The PRODIGY architecture. *Journal of Theoretical and Experimental Artificial Intelligence*, *7*(1).

Watkins, C. J. C. H., & Dayan, P. (1992). Technical note: Q-learning. *Machine Learning*, *8*(3).

Wilensky, R. (1981). Meta-planning: Representing and using knowledge about planning in problem solving and natural language understanding. *Cognitive Science*, *5*(3), 197–233.