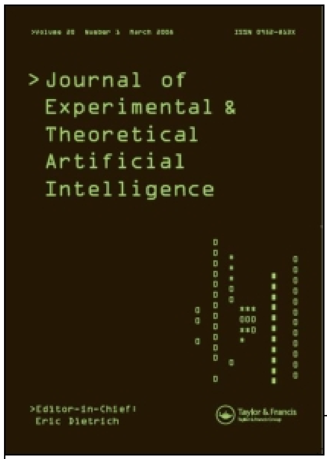
>Volume 20 Number 1 March 2008
ISSN 0952-813X
>Journal of Experimental & Theoretical Artificial Intelligence
>Editor-in-Chief: Eric Dietrich
Taylor & Francis

Journal of Experimental & Theoretical Artificial Intelligence
Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t713723652

# Meta-case-based reasoning: self-improvement through self-understanding

J. William Murdock [a]; Ashok K. Goel [b]
[a] IBM Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA
[b] Artifical Intelligence Laboratory, College of Computing, Georgia Instiute of Technology, Atlanta, GA 30332-0280, USA

Online Publication Date: 01 March 2008

To cite this Article: Murdock, J. William and Goel, Ashok K. (2008) 'Meta-case-based reasoning: self-improvement through self-understanding', Journal of Experimental & Theoretical Artificial Intelligence, 20:1, 1 — 36

To link to this article: DOI: 10.1080/09528130701472416
URL: http://dx.doi.org/10.1080/09528130701472416

*Journal of Experimental & Theoretical Artificial Intelligence*,
Vol. 20, No. 1, March 2008, 1–36

# Meta-case-based reasoning: self-improvement through self-understanding

J. WILLIAM MURDOCK*† and ASHOK K. GOEL‡

†IBM Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA
‡Artifical Intelligence Laboratory, College of Computing, Georgia Instiute of Technology, Atlanta, GA 30332-0280, USA

(*Received 14 August 2006; in final form 19 June 2007*)

The ability to adapt is a key characteristic of intelligence. In this work we investigate model-based reasoning for enabling intelligent software agents to adapt themselves as their functional requirements change incrementally. We examine 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) which supports modelling 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 that contain TMKL models of themselves.

*Keywords:* Case-based reasoning; Meta-case representation; Adaptability; Reflection

## 1. Introduction

An intelligent agent may encounter situations for which it is important to extend its range of capabilities. Over its life, changes to the requirements placed on it may concern many aspects, for example performance, constraints, resources, and functionality. In this paper, we focus on one kind of requirement change: incremental change to the functionality of the agent. Specifically, we address the addition of a new task that is related to some task that the agent already knows how to accomplish.

*Corresponding author. Email: murdockj@us.ibm.com

Case-based reasoning addresses a different problem. Given a plan to disassemble a device and the goal of disassembling another, very similar device, a case-based planner may modify ('tweak') the known plan to accomplish the new goal. In fact, this is precisely what ADDAM, a case-based agent for physical device disassembly (Goel *et al.* 1997), does. But now consider what may happen when ADDAM is given the task of generating assembly plans instead of disassembly plans. The desired assembly plan in this example has little structural similarity with any of the known disassembly plans; there is typically no overlap at all in the operators used because all the operators are inverted, and while the objects of those operators are similar, they are typically manipulated in the reverse order. A disassembly plan cannot be easily adapted into an assembly plan by simple tweaks (which explains why our experiments described later do not contain a comparison with case-based planning). The class of problems for which case-based reasoning is applicable is such that (i) the new problem is so closely related and similar to a familiar problem that the solution to the familiar problem is almost correct for the new problem, and (ii) the solution desired for the new problem is structurally so similar to the known solution to a familiar problem that the known solution needs only small modifications ('tweaks') to obtain the desired solution for the new problem. For this class of problems, in which the known solution is reused almost completely, case-based planning provides substantial computational advantages over generative planning; however, for problems outside this class, the computational complexity of plan adaptation is no better than that of generative planning (Nebel and Koehler 1995).

The key insight in the present work is that although the disassembly plans available in ADDAM's memory and the desired assembly plan have little structural similarity, the reasoning processes for producing the two kinds of plans (assembly and disassembly) are very similar. Therefore, in the present work we address 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 paper can be viewed as constituting *meta-case-based reasoning*. The class of problems for which meta-case-based reasoning is applicable is such that (a) the new task is so closely related and similar to a familiar task that the reasoning process for the familiar task is almost correct for the new task, and (b) the reasoning process required for addressing the new task is structurally so similar to the reasoning process required to address the familiar task that it needs only small modifications to obtain the desired reasoning process for the new task. These conditions can hold even if the solution to an instance of the new task is not necessarily similar to the solution to any instance of the familiar task. **Our first hypothesis is that for this class of problems, meta-case-based reasoning can provide substantial computational advantages over generating a plan for an instance of the new task.**

Like case-based reasoning, meta-case-based reasoning sets up the tasks of retrieval, adaptation, verification and storage. The work described in this paper focuses primarily on the tasks of adaptation and verification; we use an off-the-shelf knowledge representation system to provide storage and retrieval. 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 tasks may be abstract and complex; for example, the assembly task has an effect that a device is *assembled*, which is operationalized for a 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 in effects 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. **This leads to our second hypothesis: if an agent has a model of its own design (a self-model), then, for the class of problems for which meta-case-based reasoning is applicable, the agent may use the model to identify the elements in its design that need to be modified (self-adaptation) for accomplishing new tasks.**

## 2. Model-based self-adaptation

To identify part of an agent to modify, it is helpful to have knowledge about how the various parts of the agent contribute to the overall effect of that agent. The issue of identifying a candidate modification can be addressed by having a variety of specific types of transformations to accomplish specific types of differences in behaviour. Such transformations can be guided by knowledge about these behaviours. 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, in particular the following:

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

The Task–Method–Knowledge (TMK) 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 behaviour 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 uses 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 TMK Language (TMKL).

Reflective Evolutionary Mind (REM) 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 behaviour 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 partially or fully specified 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 instead to assemble a device, REM first performs an 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. However, if execution is unsuccessful, 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.

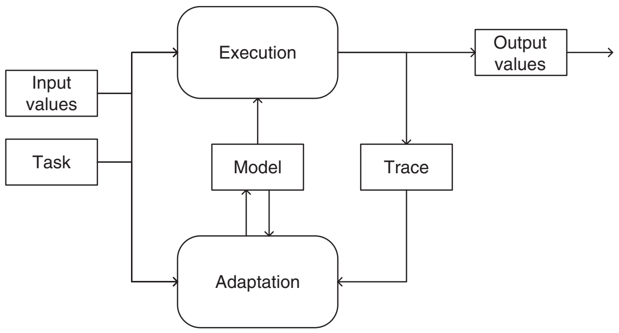

Figure 1. The architecture of REM. Large round boxes indicate procedural elements. Small boxes indicate information used and/or produced by those elements.

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).
- **Generative planning** involves determining a complete sequence of actions that accomplish the desired result. REM can use this strategy to develop a method for a new task by first building a plan for a specific parameterization of the task (e.g. disassembling a specified object) and then generalizing that plan to form a method for the task (e.g. disassembling all objects that have the same design as the original object). In addition, REM can use this strategy after an execution failure by creating and generalizing a plan for a single failed subtask.
- **Fixed-value production** involves adding a highly specialized subtask to a model that provides a single, fixed value for an output parameter in a specific situation. The main technical challenge that the algorithm addresses is *localization*: finding a subtask for which such a modification could potentially address the observed failure.
- **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.

In this paper we focus primarily on the use of relation mapping in 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, and 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 overheads 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 behaviour: (1) a language for modelling 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 modelling language to enable successful execution.

## 3. Meta-case representation

REM uses and adapts representations of reasoning processes encoding TMKL. TMKL systems are divided into tasks, methods, and knowledge. A task is a unit of computation that 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 related to 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`. More details (including examples) of all three portions of a TMKL model, are given in Murdock (2001). A relatively concise overview of this material is provided below.

## 3.1 *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 movement task 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 executed immediately. 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 include relations that are known to be external to the agent. Primitive tasks that are not actions result in 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 4 for more details on ADDAM). Performing this task removes a component from a device. The task has one input, the 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, 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). For REM to execute that task, the `do-remove` procedure must be implemented and loaded; in the ADDAM system, procedures are implemented to interact with a

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))))
```

simulated environment, but alternatively one could invoke functionality in these procedures that affect the real world and observe the consequences (e.g. through robotic arms and sensors). The parameters for the procedure match the inputs listed for the task; `do-remove` takes one parameter, a reference to the object to be removed.

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.

## 3.2 *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 starts at the first transition, goes to the state it leads to, executes the subtask for that state, selecting an outgoing transition from that state whose applicability condition holds, and then repeats the process until a terminal transition is reached.

## 3.3 *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 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 base ontology. Some are the core elements of a TMKL model: `task`, `method`, and various components of tasks and methods (e.g. `parameter`). TMKL also provides a variety of 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`, and `internal-definition-relation`. An external state relation that 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 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 provide information that 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 that 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 which can be run sequentially to produce the same effect as running neither of them are `inverse-of` each other. TMKL provides only partial formal semantics for these relations (e.g. some generic inference rules for determining whether two tasks are similar). Developers of TMKL models are expected to provide additional information about similarity and inversion that is specific to the domain in which their system operates. For example, the domain model for manufacturing described in section 4 includes assertions about inverse manufacturing actions such as screwing and unscrewing.

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 that are specific to a particular domain interchangeably. For example, one step of the Relation Mapping algorithm in section 5.2.3 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.

### 3.4 *Executing TMKL models*

The algorithm for execution in REM recursively steps through the hierarchy of tasks and methods. Given a non-primitive task, REM selects a method for that task and executes it. 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. Throughout the process, REM builds a trace of execution (to support future diagnosis and adaptation) and monitors for failures. Details of this execution process are given by Murdock (2001).

Whenever REM needs to select a method for a task or a transition within a method, it checks the stated applicability conditions of the methods or transitions. If more than one option is applicable, reinforcement learning, specifically Q-learning (Watkins and Dayan, 1992), is used to choose among those options. When the desired effect for the main task is accomplished, positive reinforcement is provided to the system, allowing it to learn to choose options that 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 are 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.

This execution algorithm is similar to a forward-chaining Hierarchical Task Network (HTN) planner (Nau *et al.* 1998, 2003). The main differences between the two is that REM's execution involves actually performing actions as they are encountered rather than just adding the actions to a plan. Consequently, REM does not perform backtracking when an action fails; instead it restarts the planning process. Furthermore, REM monitors the plan for failures in execution so that it can perform adaptation in response to failures. The primary research focus of REM has been *adaptation*, as described in the following section. The execution algorithm is (intentionally) quite simple, because we expect the power of systems encoded in REM to come primarily from a combination of powerful methods developed by users to address specific tasks and powerful adaptation strategies included in REM to improve the range of applicability of those user-developed algorithms. In contrast, HTN planners generally provide planning algorithms which are more powerful than REM's execution algorithm (e.g. including backtracking), but are generally helpless when faced with tasks that are even slightly different from the ones for which they have methods. For more about the relationship between REM and HTN planners, see section 7.1.

## 4. ADDAM: an illustrative agent example

One agent that has been modelled in TMKL and executed in REM is ADDAM (Goel *et al.* 1997). ADDAM is a physical device disassembly agent. It 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 these 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 be actually useful for automation, a disassembly plan do needs ultimately to be translated into robotic movements. ADDAM has a separate module which 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 included this module 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 seem to be 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 approach that ADDAM takes to this problem is case-based reasoning; it adapts old plans for disassembling devices into new plans for disassembling similar devices.

## 4.1 *ADDAM knowledge*

The representation of devices in ADDAM describes not only individual components and connections but also more abstract subassemblies which 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 for disassembling the storage subsystem. That node has children for 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 that 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, and 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 that ADDAM is able to disassemble is a hypothetical layered roof design with 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 and o 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 first creates a new disassembly plan and then executes 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 constructs a mapping between elements of the new and existing devices, converts portions of the plan for the existing device into analogous portions of the plan for the new device, and then converts the ordering dependencies for the existing plan into analogous ordering dependencies for the new plan.

## 5. Meta-case-based reasoning process

If a user asks REM to perform a task, such as assembling a device, REM first checks to see whether that task already has an existing implementation (i.e. a method or a primitive implementation). If it does not, then REM needs to employ its meta-case-based reasoning process to adapt its existing knowledge to satisfy the requirements of the task. That process involves (a) retrieving an existing task that has a method and is similar to the desired task, (b) adapting a method for the retrieved task to implement the desired task, (c) verifying that the adapted method does accomplish the desired effect by executing the desired task with that method, and (d) storing the adapted method for future use when the same task is encountered and for future adaptation when another similar task is encountered.

### 5.1 *Retrieval*

As described in section 3.3, REM uses Loom as its mechanism for storing and accessing knowledge. To retrieve an existing task that is similar to some specified task, REM issues a Loom query asking for any task that has a method for which the `similar-to relation` holds with the specified task.

Loom decides whether a relation holds in two ways: it may have an explicit assertion in memory indicating that the relation holds, or it may use an inference rule (i.e. a statement that some condition `implies` that the relation holds). As noted in section 3.3, REM includes some domain-independent rules for inferring the `similar-to` relation. For example, REM has a rule which states that tasks which have similar effects are similar to each other. It determines that tasks have similar effects if they meet the following requirements:

- they have the same sets of parameters,
- **and** they have similar `given` conditions,

- **and** they have similar `makes` conditions.

Conditions (i.e. logical expressions encoded in the Loom syntax) condition are identified as similar if they meet the following requirements:

- they are equal,
- **or** they both have the form (`<operator>` `<subexpression>` ...), **and** the operators (e.g. :and, :or, :not) are identical, **and** each sub-expression in one is similar to the corresponding subexpression in the other one,
- **or** they both have the form (`<predicate>` `<term>` ...), **and** the predicates are identical, **and** each term in one is similar to the corresponding term in the other one.

Terms within a condition are identified as similar if they meet the following requirements:

- they are equal,
- **or** they are formula of the form (`<relation>` `<term>` ...) **and** the relations are identical, **and** each term in one is similar to the corresponding term in the other one,
- **or** they are references to concrete instances, **and** those concrete instances are similar.

Similarity of concrete instances is generally encoded in the knowledge portion of a particular TMKL model or input knowledge state. Instances may be asserted to be similar via direct assertions (e.g. a particular problem specification may say that `board-1` is similar to `board-2`), or similarity may be derived from domain-specific inference rules (e.g. a model may indicate that all screws are similar to each other).

The simple domain-independent rules for task similarity (grounded in domain-specific rules for instance similarity) are sufficient for all the retrieval that was performed in the experiments described in this paper. However, in other contexts, recognizing similarity among tasks may require more specialized rules. In those cases, TMKL model developers are expected to add direct assertions and/or additional inference rules defining similarity of tasks in the domain of interest.

## 5.2 *Adaptation*

There are four general varieties of adaptation that 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.

Fixed-value production has been described in detail elsewhere (Murdock and Goel 1999a) and is not used in the ADDAM example. The remaining adaptation strategies are described below.

**5.2.1 Situated learning adaptation.** If REM is given a description of a task, some inputs, and a description of available actions, it must be able to generate a method 'from scratch', i.e. without having any existing method to modify. The situated learning strategy addresses this problem by pure Q-learning, i.e. it simply tries arbitrary actions and sees what happens. The term 'situated' is intended to indicate that behaviour is guided entirely by action and experience and does not involve explicit planning or deliberation. The situated learning mechanism in REM creates a method which 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, 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))` 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, and so all the decision making required for the process is deferred until execution time.

**5.2.2 Generative planning adaptation.** Like situated learning, generative planning is also applicable even if there is no existing method to modify. Its applicability conditions are more restrictive than those of situated learning. Specifically, generative planning requires that the effects of each task be known and be expressible in the formalism employed by a generative planner. In contrast, situated learning can be accomplished even when effects can only be observed by trying out the actions in the real world or in a (potentially complex and/or hidden) simulator. Situated learning can even be employed in situations in which effects are only partially observed and/or may only be apparent after several more actions as long as the agent can eventually discover whether a task has been successfully performed. However, situated learning can be extremely expensive (as shown in section 6.1), especially if feedback about results is delayed.

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© (Blum and Furst 1997) as its external planner. The generative planning mechanism in REM can be used not only before execution to construct a method for a novel main task, but also after a failure to construct a method for a faulty subtask. The code for Graphplan (© 1995, A. Blum and M. Furst) was downloaded from `http://www.cs.cmu.edu/afs/cs.cmu.edu/usr/avrim/Planning/Graphplan/`. 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 paper 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 that 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 post-conditions for Graphplan operators. 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 than 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 that has not been used in an earlier action and is an input parameter to the main task, the input parameter to the main task is linked to the corresponding parameter in the primitive action. Additional references to the same value are translated into 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 *et al.* 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 which require such knowledge (e.g. Hierarchical Task Network planners) are not appropriate for this mechanism.

**5.2.3 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). For 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 that 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

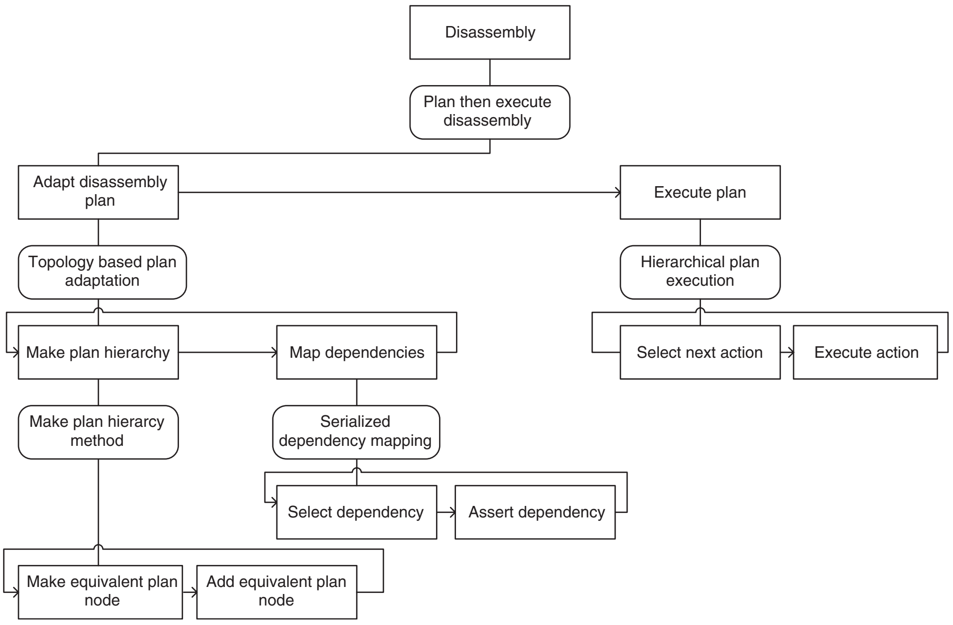

Figure 2. A diagram of some of the tasks and methods of ADDAM.

into two major portions: `Make Plan Hierarchy` and `Map Dependencies`. `Make Plan Hierarchy` involves constructing plan steps, e.g. unscrew `Screw-2-1` from `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 for example, the two boards must be unscrewed before they can be removed. The heart of the dependency mapping process is the selection of a potential ordering dependency and the assertion of that dependency for that plan. Note that dependencies are much simpler than plan nodes; a dependency is just a binary relation while a node involves an action type, some objects, and information about its position in the hierarchy. The relative simplicity of dependencies is reflected in the implementation of the primitive task that asserts them; this task is implemented by a simple logical assertion (in the task's `:asserts slot`) which states 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 that 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 techniques described here (situated learning, generative planning, and relation mapping) can be used. Because the model for ADDAM includes its primitive actions, the two kinds of adaptation that 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 that 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 inputs to the process include a main task, an implemented task similar to the main task, and a knowledge state in which the main task should be performed. For example, it could receive an unimplemented task of assembly, an implemented task of disassembly, and a state in which some components are available but not connected to each other. The similar task was retrieved earlier in the meta-case-based reasoning process via a Loom `similar-to` query. In the assembly example, there is no user-supplied assertion that assembly and disassembly are similar; rather, the similarity is computed using the rules described in section 5.1.

The relation mapping algorithm starts by copying the methods (including the methods' subtasks and those subtasks' methods) for `known-task` and asserting that these copied methods are 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 that 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

Table 2. Algorithm for relation mapping.

```
Algorithm relation mapping(main-task)
Inputs:      main-task: The task to be adapted
             known-task: A task such that (similar-to main-task known-task) holds
             current-knowledge-state: The knowledge to be used by the task
Outputs:     main-task: The same task with a method added
Effects:     A new method has been created that accomplishes main-task within
             current-knowledge-state using a modified version a method for known-task.

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
```

used is `inverse-of`. Other experiments have employed the relation-mapping algorithm with other relations, such as generalization and specialization (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 that 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` that indicate ordering relations among nodes in a hierarchical plan; these relations are the inverse of each other and 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 inverses 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 that 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 that 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 that involve these relations and to modify those tasks accordingly. For each task, REM determines if the effect on the relation is implemented by a direct logical assertion. If so, REM can simply change that assertion by applying the mapping relation. If not, REM cannot alter the task itself and must introduce a separate *mapping task* to apply the mapping relation to the relevant manipulable relation.

Figure 3 presents the results of the relation mapping process over ADDAM in the assembly example, again focusing on those elements that are relevant to this discussion. After the disassembly process is copied, three major changes are made to the copied version of the model.

(i) The `Make Plan Hierarchy` process is modified to adjust the type of actions produced. Because the primitive tasks that manipulate plan nodes (e.g. `Make Equivalent Plan Node` in table 3) are implemented by procedures, REM is not able to modify them directly. 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.

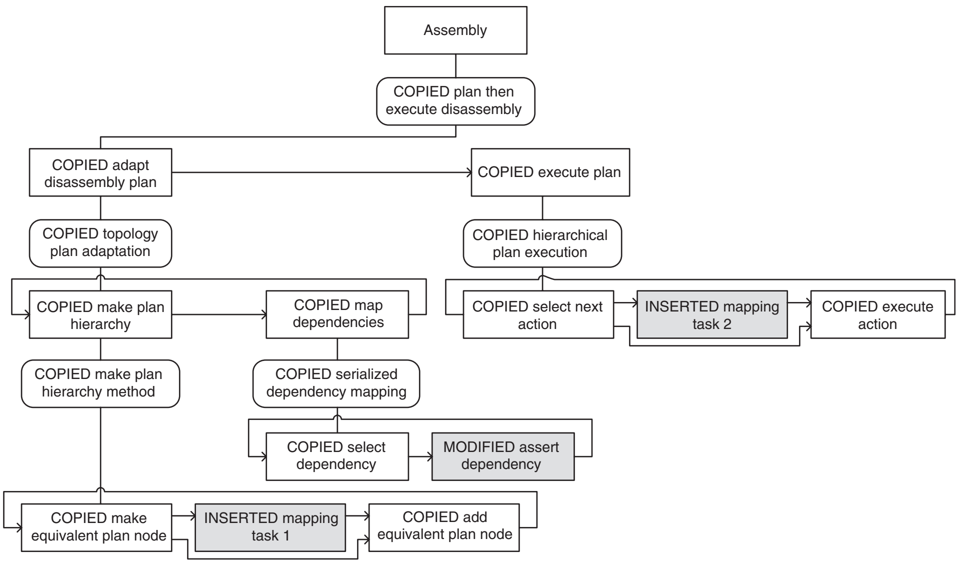
Assembly
COPIED plan then execute disassembly
COPIED adapt disassembly plan
COPIED execute plan
COPIED topology plan adaptation
COPIED hierarchical plan execution
COPIED make plan hierarchy
COPIED map dependencies
COPIED select next action
INSERTED mapping task 2
COPIED execute action
COPIED make plan hierarchy method
COPIED serialized dependency mapping
COPIED select dependency
MODIFIED assert dependency
COPIED make equivalent plan node
INSERTED mapping task 1
COPIED add equivalent plan node

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

(ii) 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 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. Table 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 `before` and `after` relations). For example,

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)))
```

if an old disassembly plan requires that boards be unscrewed *before* they can be removed, the new assembly plan will require that they be screwed together *after* they are placed.

(iii) The plan execution process is modified to adjust the type of actions executed. This adjustment is done by an inserted task that 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 deduce analytically from the model that it was inverting the same actions twice. However, the model does not contain the level of detail required to deduce that the actions being produced in the early portion of the process are the same ones being executed in the later portion. Even if the information were there, it would be in the form of logical expressions about the requirements and results of all of the intervening tasks (which are moderately numerous, since these inversions take place in greatly separated portions of the system); reasoning about whether a particular knowledge item were being inverted twice for this problem would be a form of theorem proving over a large number of complex expressions, which can frequently be intractable.

Fortunately, it is not necessary for REM's model-based adaptation technique to prove deductively 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 that goes into the inserted task and one that goes around it. During the verification step (described in the next section), the decision-making (Q-learning) process selects between these two transitions. Through experience, the decision-making process develops a policy of including one of the inserted mapping tasks but not both.

## 5.3 *Verification*

Once REM has completed the adaptation, it employs the TMKL execution process described in section 3.4 to accomplish the task that the user requested. If the result requested by the task has been accomplished, then REM has verified that the adaptation was successful. If not, REM must make additional modifications and attempt the task again.

If adaptation resulted in a model with multiple execution options, the additional modifications can be as simple as applying negative feedback to the reinforcement learning mechanism. In the ADDAM assembly example, REM decides during execution whether to perform each of the optional inserted mapping tasks. Note that the decision-making mechanism used in this example is the same one used in situated learning; however, here this component is being used *only* to decide among the options that model-based adaptation left unspecified. In contrast, the situated learning algorithm uses 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 itself, is complex); this fact is strongly reflected in the results presented in section 6.1.

The use of reinforcement learning to address unresolved redesign decisions makes REM substantially less brittle than it would be with an entirely rule-based adaptation process. Some domains may be superficially similar but demand different solutions. For example, some planning domains may have an `inverse-of` relation such that plans can be inverted by doing exactly the same actions in the inverse order. In those domains, a correct adaptation would involve inverting the order of actions but not inverting the action types. The existing REM system could handle such a domain without any modification; the Relation Mapping algorithm would still insert the same optional tasks for applying the `inverse-of` relation to action types, but the reinforcement learning mechanism would then learn *not* to use these optional tasks.

If reinforcement learning results in a policy that allows the task to complete successfully (as defined by its `makes` condition), the adaptation has been verified. Once verified, the adapted method for the task can be stored for future use.

## 5.4 *Storage*

Since the adapted model is encoded in Loom, it is immediately available for retrieval in response to a future task (as described in section 5.l). Loom also provides facilities for storing and loading a knowledge base to and from a file. This makes it possible to store REM's case library and reuse it in a future session.

REM does not attempt to do any special optimization of TMKL model storage beyond what Loom provides. In future work, it may be useful to build explicit indices over those portions of the TMKL model that are frequently used to judge similarity (e.g. the `given` and `makes` conditions of the main task); we would expect that this would allow significantly faster access over large repositories of models than a general-purpose knowledge representation system can provide. For a very large library of models, it would be infeasible to load all of the known models into memory at once; in that case effective indexing would be essential (e.g. using a relational or object-oriented database).

In our experience, highly optimized indexing is less crucial for meta-case-based reasoning than it is for traditional case-based reasoning. In case-based reasoning, a new case is generated for every distinct task instance performed (e.g. each device that ADDAM disassembles results in a seperate case); thus the library of stored cases gets large very quickly. In contrast, meta-cases (e.g. moving from disassembly to assembly) seem to be much less frequent in practice. However, we believe that there are domains in which new tasks are moderately frequent and do accumulate over long periods of time. For example, an sufficiently adaptive world wide web agent may be able to operate for years; to do so, it would need to accumulate many meta-cases to react to the many changes in the tasks that are applicable to the web. Thus improvements to meta-case storage appear to be a useful topic for future work.

## 6. Evaluation

Evaluation of REM has involved a variety of experiments and theoretical analysis. Much of this work has involved 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 a variety of devices. Some of these devices include a disposable camera, a computer, etc. The nested roof example discussed in section 4 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 required by REM for 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 the 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. 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 because of memory management problems either with Graphplan or with REM's use of it. 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

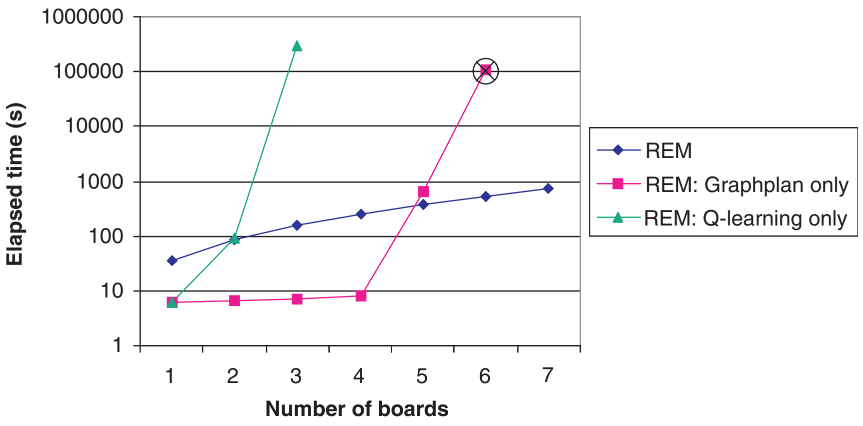

Figure 4. Logarithmic scale graph of the relative performances of different techniques within REM on the roof assembly example (which involves many goal conflicts) for a varying number of boards. The 'X' through the last point on the Graphplan line indicates abnormal termination (see text).

boards; in contrast, REM's relation mapping (with assistance from Q-learning) shows relatively steady performance.

The reason for the steady performance using relation mapping 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 being assembled (in this case, the roof) 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 are generative planning or reinforcement learning techniques.

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 as the roof designs in the previous problem. However, planning assembly for these roofs using generative planning is much easier than for those 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 performance of the different approaches in this experiment. In this experiment, REM using only Graphplan is able to outperform the model transfer approach; 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: 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.

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

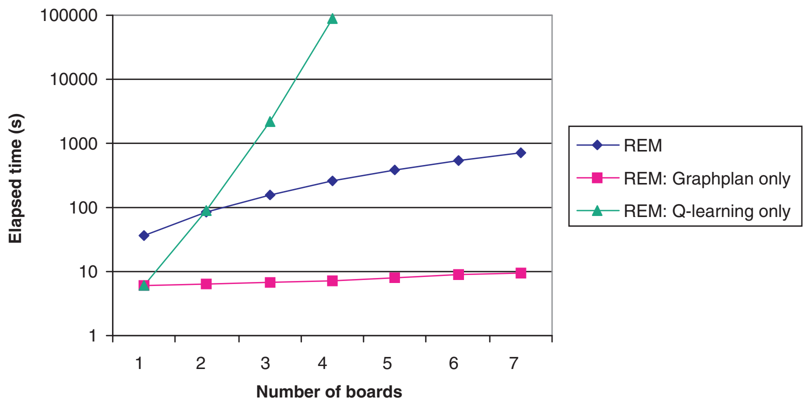

Figure 5. Logarithmic scale graph of the relative performances of different techniques within REM on a modified roof assembly example that has no conflicting goals.

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 containing substantially more components. However, there are other, similar devices involving 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); however, our domain has 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, and can only be placed in one of three locations (the warehouse, a loading pallet, and the truck); this loading 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 given by Nilsson (1998, errata, p. 4). These two methods are redundant; the behaviour of `Nilsson's Strategy` when there is only one crate is identical to the behaviour 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 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; (iii) REM with no model used generative planning to solve the entire problem. The results of this experiment (Murdock and Goel 2003) are consistent with the results in the ADDAM assembly problems: for harder problems, adaptation of existing hierarchies of tasks and methods provides a substantial advantage over generative planning (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 10 seconds.
- REM with the ablated model took approximately 1 hour.
- REM with no model took approximately 4.5 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 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 a more specific or more 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 to 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 adapt reflectively, it would be forced to solve the entire problem using generative planning or situated learning. 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 behaviour.

### 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 behaviour of Mosaic 2.4 (Murddock and Goel 1999b). 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 adaptation (using model and trace information) enables REM to localize the portion of the browser that needs to be modified and to make the appropriate modification.

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 and Geol 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 summarize the conclusions that 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 paper (Murdock 2001).

- The cost of execution (both before and after adaptation) is potentially unlimited because an arbitrary agent may be arbitrarily expensive to run. However, it is possible to analyse 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 $t$, the 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 that 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 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 and experimental results forms the evaluation for the work presented in this paper. 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 that are computationally expensive for techniques that do not employ a model.

## 7. Related research

### 7.1 *Generative planning*

Generative planning (Tate *et al.* 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 and Gurst 1997, Hoffmann and Nebel 2001) have provided systems that 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 different, 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. Thus the enormous cost of constructing a plan must be incurred over and over.

Generative planning and 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, model-based adaptation alone 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 that 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 using 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.

One approach to hierarchical planning that is particularly closely related to this research is Hierarchical Task Network (HTN) planning (e.g. Ero *et al.* 1994). Models in TMKL resemble HTNs in that both involve tasks that are decomposed by methods into partially ordered sets of subtasks. REM uses TMKL for modelling and executing reasoning processes, while HTNs are typically used for planning. There are two other key distinctions between the TMKL and HTN formalisms.

- 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). HTNs 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 from the beginning whenever it reaches a failed state, acceptable performance is only possible when there are very few unspecified decision points.
- TMKL encodes information in tasks and methods that is not directly needed for performing those tasks and methods but 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, HTNs are primarily used in systems which assume that a complete and correct HTN exists and only attempt tasks encoded in the HTN. When automatic learning of HTNs is done (Ilghami *et al.* 2002), existing HTNs 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 (Hamnmond 1989). As described earlier, ADDAM is a case-based planner. In contrast, REM's model-based adaptation techniques constitute 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 very similar to the desired plan and the retrieval method can make use of domain-specific plan indexing (Nebel and 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. For example, KRITIK, a case-based system for designing physical devices, uses a structure–behaviour–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 and Kenne 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 and Ullrich 1999). This can provide some of the efficiency benefits of case-based planning and still keep the breadth of coverage of generative planning. However, it still suffers from the cost problems of generative planning when cases are not available and the limited nature of plan adaptation when they are available.

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 that 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 (Velosos 1994) or improving plan quality via explanation-based learning (Pérez and Carbonell 1994). However, these techniques assume that there is a single underlying reasoning process (the generative planning algorithm) and focus on fine tuning that process. This is very effective for problems that are already well suited to generative planning, making them even better suited to planning over time as the planner becomes tuned to the domain. However, it does not address problems that 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 *et al.* 1995) considers the reuse of adaptation processes within case-based reasoning. Case-based adaptation does not use complex models of reasoning because it restricts its reasoning about processes to a single portion of case-based reasoning (adaptation) and assumes a single strategy (rule-based search). This limits the applicability of the approach to the (admittedly quite large) set of problems for which adaptation by rule-based search can be effectively reused. It does provide an advantage over our approach in that it does not require a functional model of representation of the other portions of the case-based reasoning process. However, given 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.

The output of a case-based planner is a plan to address a specific goal from a specific starting state. The output of meta-case-based reasoning is a new reasoning process which can potentially produce plans for a variety of starting states. Once a meta-case-based reasoning process for generating disassembly plans is adapted to generate assembly plans, REM can directly invoke its method for generating an assembly plan each time a new assembly goal is presented (and can still 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 ADDAM example described in detail in this paper shows meta-case-based reasoning over a case-based planning process, but the other examples discussed in sections 6.2 and 6.3 involve REM adapting other sorts of processes.

## 7.3 *Machine learning*

Reinforcement learning (Kaelblig *et al.* 1996) is one popular machine learning technique. An agent operating by reinforcement learning alone simply tries out actions, observes the consequences, and eventually begins to favour those actions that tend to lead to desirable consequences. However, a major drawback of this approach 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, and 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 robot assembly plans (Segre 1988). More recently, DerSNLP (Ihrig and Kambhamopati 1997) has used EBG for repairing plans and learning planning knowledge. As described in section 5.2.2, REM also 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 these systems perform planning in the context of extensive background knowledge about the domain and the available reasoning mechanisms. MOLGEN divides this background knowledge 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 a 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 includes 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. However, unlike PAM and MOLGEN, 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 contains 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 three 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 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 that has been tested or proven to be correct. It is not acceptable for Guardian just to 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 behaviour. In contrast, the web browsing domain, for example, not only allows an agent to try out new kinds of behaviours 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 required by Guardian's domain. 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 which uses meta-rules to guide the ordering of inference steps. Anytime algorithms (Boddy and Dean 1989) perform explicit deliberation-scheduling to allocate processing time to a set of algorithms which make incremental enhancements to an existing solution to a problem. Similarly, the Rational Meta-Reasoning approach (Russell and Wefald 1991, chapter 3) computes the expected benefits of applying algorithms versus simply acting. 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 modelling*

The field of agent modelling provides an important perspective on the question of what information provides a useful representation of a process. The primary difference between the work described in this paper and existing work on agent modelling is that our work uses agent models for automated self-adaptation to perform previously unknown tasks. Most other agent modelling projects focus on semi-automated processes. For example, CommonKADS (Schreiber *et al.* 2000) and DESIRE (Brazier *et al.* 1997) are methodologies for building knowledge systems that use models of agents throughout the development process.

TMKL is particularly closely related to earlier TMK formalisms (Goel and Mordock 1996, Griffith and Murdock 1998, Murdock and Goel 2001) and Generic Tasks (Chandrasekaran 1986). A particularly significant ancestor of TMKL is the SBF-TMK language in Autognostic (Stroulia and Geol 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 and Leake 1995) uses an intentional model for refining case indexing more directly. In contrast, CASTLE (Freed *et al.* 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 types of adaptation can be alternatively seen as repairing a failure (as in Autognostic) or adding new capability (as in REM). This difference in perspective is 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.

Autognostic and REM use different formalizations of TMK. REM's TMKL provides a 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 5.2.3 provides a particularly substantial example of how knowledge of the domain can be used in adaptation.

TMKL also provides a more elaborate account of how primitive tasks are represented than the SBF-TMK formalism in Autognostic does. Primitive tasks in Autognostic always include a link to a LISP procedure that 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, but, like Autognostic, it is unable to modify these procedures. However, TMKL also allows other types 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 types of primitive functionality may not be adequately represented by these sorts of primitives in which case they must be implemented 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 that 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 which represents a novel compromise in this three-dimensional space. In particular, we have described a model-based technique for agent self-adaptation when (a) the new task is so closely related and similar to a familiar task that the reasoning process for the familiar task is almost correct for the new task, and (b) the reasoning process required for addressing the new task is structurally so similar to the reasoning process required to address the familiar task that it needs only small modifications to obtain the desired reasoning process for the new task. These conditions may apply even if the *solutions* to the new task are radically different from the solutions to the old task.

The model-based technique for self-adaptation described above is computationally efficient, has reasonable knowledge requirements, and provides 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 are generally available to the designers of the agents. The model-based technique requires only that the agents have these models and provides the TMKL modelling language for encoding the models. However, at present the model-based technique can only address small, incremental changes to the functional requirements of the agent; 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 such as inversion, generalization, or specialization.

For simple adaptation problems, the model-based technique successfully identifies both the design elements that need to be changed and the changes needed. 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

G. Abowd, A.K. Goel, D.F. Jerding, *et al.*, "MORALE: mission oriented architectural legacy evolution", in *Proceedings of the International Conference on Software Maintenance (ICSM-97)*, 1997.

A. Blum and M.L. Furst, "Fast planning through planning graph analysis", *Aritif. Intell.*, 90, pp. 281–300, 1997.

M. Boddy and T. Dean, "Solving time-dependent planning problems", in *Proceedings of the 11th International Joint Conference on Artificial Intelligence IJCAI-89*, San Francisco, CA: Morgan Kaufmann, 1989, pp. 979—984.

F.M.T. Brazier, B.D. Keplicz, N. Jennings, and J. Treur, "DESIRE: modelling multi-agent systems in a compositional formal framework", *Inter. J. Coop. Inform. Systems*, 6, pp. 67–94, 1997.

B. Chandrasekaran, ''Generic tasks in knowledge-based reasoning: high-level building blocks for expert systems design'', *IEEE Expert*, 1, pp. 23–30, 1986.

K. Erol, J. Hendler, and D.S. Nau, ''HTN planning: complexity and expressivity'', in *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, Cambridge, MA: AAAI Press, 1994.

S. Fox and D.B. Leake, ''Using introspective reasoning to refine indexing'', in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, 1995, pp. 391–399.

M. Freed, B. Krulwich, L. Birnbaum, and G. Collins, ''Reasoning about performance intentions'', in *Proceedings of 14th Annual Conference of the Cognitive Science Society*, Hillsdale, NJ, 1992, pp. 7–12.

M.R. Genesereth, ''An overview of meta level architectures'', in *Proceedings of the 3rd National Conference on Artificial Intelligence (AAAI-83)*, 1983, pp. 119–124.

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

A.K. Goel and J.W. Murdock, ''Meta-cases: explaining case-based reasoning'', in *Proceedings of the 3rd European Workshop on Case-Based Reasoning -(EWCBR-96)*, I. Smith and B. Faltings, eds, Berlin: Springer-Verlag, 1996.

T. Griffth and J.W. Murdock, ''The role of reflection in scientific exploration'', in *Proceedings of the 20th Annual Conference of the Cognitive Science Society*, 1998.

K.J. Hammond, *Case-Based Planning: Viewing Planning as a Memory Task*. New York: Academic Press, 1989.

B. Hayes-Roth, ''An architecture for adaptive intelligent systems'', *Artif. Intell.*, 72, pp. 329–365, 1995.

J. Hoffmann and B. Nebel, ''The FF planning system: fast plan generation through heuristic search'', *J. Artif. Intell. Res.*, 14, pp. 253–302, 2001.

L.H. Ihrig and S. Kambhampati, ''Storing and indexing plan derivations through explanation-based analysis of retrieval failures'', *J. Artif. Intell. Res.*, 7, pp. 161–198, 1997.

O. Ilghami, D. Nau, H. Muñoz-Avila, and D.W. Aha, ''CaMeL: learning methods for HTN planning'', in *Proceedings of the Sixth International Conference on AI Planning and Scheduling (AIPS-02)*, Cambridge, MA: AAAI Press, 2002

International Planning Competition. ''Competition domains'', Available online at: http://planning.cis.strath.ac.uk/competition/domains.html, 2002 (accessed March 2006).

L.P. Kaelbling, M.L. Littman, and A.W. Moore, ''Reinforcement learning: A survey'', *J. Artif. Intell. Res.*, 4, 1996.

C.A. Knoblock, ''Automatically generating abstractions for planning'', *Artif. Intell.*, 68, pp. 243–302, 1994.

D.B. Leake, A. Kinley, and D. Wilson, ''Learning to improve case adaptation by introspective reasoning and CBR'', in *Proceedings of the 1st International Conference on Case-Based Reasoning (ICCBR-95)*, 1995.

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

R. MacGregor, ''Retrospective on Loom'', Available online at: http://www.isi.edu/isd/LOOM/papers/macgregor/Loom_Retrospective.html (accessed August 1999).

E. Melis and C. Ullrich, ''Flexibly interleaving processes'', in *Proceedings of the 3rd International Conference on Case-Based Reasoning (ICCBR-99)*, 1999, pp. 263–275.

T.M. Mitchell, R. Keller, and S. Kedar-Cabelli, ''Explanation-based generalization: A unifying view'', *Mach. Learning*, 1, pp, 47–80, 1986.

J.W. Murdock and A.K. Goel, ''Localizing planning with functional process models'', in *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS-03)*, 2003.

J.W. Murdock, ''Self-Improvement through Self-Understanding: Model-Based Reflection for Agent Adaptation'', PhD thesis, Georgia Institute of Technology, College of Computing, 2001, Avaliabvle online at: http://thesis.murdocks.org.

J.W. Murdock and A.K. Goel, ''An adaptive meeting scheduling agent'', in *Proceedings of the 1st Asia-Pacific Conference on Intelligent Agent Technology (IAT-99)*, 1999a.

J.W. Murdock and A.K. Goel, ''Towards adaptive web agents'', in *Proceedings of the Fourteenth IEEE International Conference on Automated Software Engineering (ASE-99)*, 1999b.

J.W. Murdock and A.K. Goelm, ''Learning about constraints by reflection'', in *Proceedings of the Fourteenth Canadian Conference on Artificial Intelligence (AI-01)*, E. Stroulia and S. Matwin, Eds, 2001, pp. 131–140.

D.S. Nau, T.C. Au, O. Ilghami, et al., ''SHOP2: an HTN planning system'', *J. Artif. Intell. Res.*, 20, pp. 379–404, 2003.

D.S. Nau, S.J.J. Smith, and K. Erol, ''Control strategies in HTN planning: Theory versus practice'', in *Proceedings of the Tenth Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, 1998, pp. 1127–1133.

B. Nebel and J. Koehler, "Plan reuse versus plan generation: a theoretical and empirical analysis", *Artif. Intell.*, 76, pp. 427–454, 1995.

N.J. Nilsson, "Artificial Intelligence: A New Synthesis, San Francisco, CA: Morgan Kaufmann, 1998. Errata available online at: http://www.mkp.com/nils/clarified. (accessed May 2001).

M.A. Péerez and J.G. Carbonell, "Control knowledge to improve plan quality", in *Proceedings of the Second International Conference on AI Planning Systems (AIPS-94)*,1994.

S. Russell and E. Wefald, *Do the Right Thing: Studies in Limited Rationality*, Cambridge, MA: MIT Press, 1991.

E.D. Sacerdoti, "Planning in a hierarchy of abstraction spaces", *Artif. Intell*, 5, pp. 115–135, 1974.

G. Schreiber, H. Akkermans, A. Anjewierden, *et al.*, *Knowledge Engineering and Management: The CommonKADS Methodology*, Cambridge, MA: MIT Press, 2000.

A.M. Segre, *Machine Learning of Robot Assembly Plans*. Durdrecht: Kluwer, 1988.

B. Smyth and M.T. Keane, "Adaptation-guided retrieval: Questioning the similarity assumption in reasoning", *Artif. Intell.*, 102, pp. 249–293, 1998.

M. Stefik, "Planning and meta-planning (MOLGEN: Part 2)", *Artif. Intell.*, 16, pp. 141–169, 1981.

E. Stroulia and A.K. Goel, "Functional representation and reasoning in reective systems", *J. Appl. Intell.*, 9, pp. 101–124, 1995.

A. Tate, J. Hendler, and M. Drummond, "A review of AI planning techniques", in *Readings in Planning*, J. Allen, J. Hendler, and A. Tate, eds, 1990, pp, 26–47.

M. Veloso, "PRODIGY/ANALOGY: analogical reasoning in general problem solving", in *Topics in Case-Based Reasoning*, Berlin: Springer Verlag, 1994, pp. 3–50.

M. Veloso, J. Carbonell, A. Péerez, D. Borrajo, E. Fink, and J. Blythe, "Integrating planning and learning: The PRODIGY architecture", *J. Exp. Theor. Artifi. Intell.*, 7, pp. 81–120, 1995.

Christopher J.C.H. Watkins and P. Dayan, "Technical note: Q-learning", *Mach. Learning*, 8, pp. 279–292, 1992.

R. Wilensky, "Meta-planning: representing and using knowledge about planning in problem solving and natural language understanding", *Cognitive Science*, 5, pp. 197–233, 1981.