# Meta-case-Based Reasoning: Using Functional Models to Adapt Case-Based Agents

J. William Murdock and Ashok K. Goel

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
{murdock,goel}@cc.gatech.edu

**Abstract.** It is useful for an intelligent software agent to be able to adapt to new demands from an environment. Such adaptation can be viewed as a redesign problem; an agent has some original functionality but the environment demands an agent with a slightly different functionality, so the agent redesigns itself. It is possible to take a case-based approach to this redesign task. Furthermore, one class of agents which can be amenable to redesign of this sort is case-based reasoners. These facts suggest the notion of "meta-case-based reasoning," i.e., the application of case-based redesign techniques to the problem of adapting a case-based reasoning process. Of course, meta-case-based reasoning is a very broad topic. In this paper we focus on a more specific issue within meta-case-based reasoning: balancing the use of relatively efficient but knowledge intensive symbolic techniques with relatively flexible but computationally costly numerical techniques. In particular, we propose a mechanism whereby qualitative functional models are used to efficiently propose a set of design alternatives to specific elements within a meta-case and then reinforcement learning is used to select among these alternatives. We describe an experiment in which this mechanism is applied to a case-based disassembly agent. The results of this experiment show that the combination of model-based adaptation and reinforcement learning can address meta-case-based reasoning problems which are not effectively addressed by either approach in isolation.

## 1 Issues

Case-based reasoning systems are inherently flexible. As long as a new problem sufficiently resembles an old problem, the case-based reasoner can address that problem. Furthermore, case-based reasoners tend to become increasingly flexible over time. As they develop experience, they build an increasingly large case library and thus may have an increasingly broad range of similar problems that they are able to address. However, some kinds of environments demand an additional kind of flexibility: the ability to reason not only about the cases but also about the case-based reasoning processes themselves.

Consider an intelligent agent which disassembles physical devices using case-based reasoning to adapt disassembly plans. When it is asked to disassemble a

D.W. Aha and I. Watson (Eds.): ICCBR 2001, LNAI 2080, pp. 407–421, 2001.

new device, it simply retrieves an existing plan to disassemble a similar device, adapts that plan, and then executes it. However, if such an agent is given a new sort of request, it is likely to be completely unsuccessful at satisfying that request. For example, if the disassembly system is asked to assemble a device instead of disassembling it, that system is unlikely to be able to address the problem at all; some or all of its processes for plan retrieval, adaptation, execution, etc. are likely to be specifically constructed for disassembly and thus should be unsuited to the task of assembly. However, the necessary processes for assembly may be similar to those which are used for disassembly. Thus it may be possible to adapt the disassembly processes so that they can be used for assembly.

There is some work which has shown that adaptation of reasoning can be facilitated by a functional model that specifies how the elements of an agent's design combine to achieve its functions [4,13,14]. Furthermore, models of this sort have been shown to be useful for encoding "meta-cases" [8], i.e., cases of case-based reasoning. Unfortunately, existing mechanisms for adaptation using functional models are effective for only a limited range of problems [12]. In particular, this approach can address those problems for which there is an existing strategy that solves a similar problem and the functional differences in the problems are particularly simple. If the functional differences are complex, then the information in the model may not be sufficient to completely determine all of the characteristics of the final solution. There are less knowledge intensive reasoning techniques which can be employed to address issues which model-based adaptation leaves unresolved. Of course, less knowledge intensive techniques tend to be more computationally costly, particularly for large problems. However, if the knowledge in the models is thorough enough to provide large pieces of the solution and to identify which elements of the solution still need to be resolved, then the portion of the problem left for a less knowledge intensive technique may be much smaller. Completing the solution can thus provide relatively small challenges to sub-symbolic reasoning techniques such as reinforcement learning and thus may be tractably completed by these techniques even when the complete problems themselves cannot be.

## 2 Functional Models as Meta-cases

A *model* is an explicit representation of a phenomenon which supports inferences about both static and dynamic properties of that phenomenon. A *functional model* is a model in which function, i.e., the intended effect of a system, plays a central role. For example, functional models of physical devices are useful for representing and organizing design cases and supporting case retrieval and adaptation [5,7]. The use of functional models as meta-cases requires models that encode abstract, computational devices, i.e., reasoning processes. These models of reasoning processes can serve an analogous role in meta-case-based reasoning to the role that is served by models of physical devices in design reasoning.

One approach to functional modeling of reasoning processes is TMK (Task Method Knowledge). TMK models provide information about the function of

agents and their elements (i.e., the tasks that they address) and the behavior of those agents 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). TMK has been used to model many types of reasoning processes, including not only case-based reasoning [8], but also scheduling using search [12], scientific discovery using analogy [9], and many others. In this paper we concentrate on the use of TMK in the context of case-based reasoning.

We have built a reasoning shell, called REM (Reflective Evolutionary Mind), which provides capabilities for executing and adapting agents represented by TMK models. The specific implementation of the TMK modeling approach encoded in REM is called TMKL. Like all variations of TMK, TMKL breaks down into three major portions: tasks, methods, and knowledge. TMKL instantiates these three portions in the following ways:

**Tasks:** Tasks are functional elements: a description of a task encodes what that piece of computation is intended to do. A task in TMKL is described by the kinds of knowledge used as inputs and outputs as well as logical assertions which are required to hold before and after the task executes. The assertions involve concepts and relations, which are defined in the knowledge portion of the TMKL model. In addition, some tasks contain information about how they are implemented. TMKL allows three types of tasks, categorized by the information that they contain about implementation:

- **Non-primitive tasks** have a slot which contains a list of methods which can accomplish that task.
- **Primitive tasks** have some direct representation of the effects of the task. TMKL allows several different ways to specify primitive tasks including links to LISP procedures which accomplish the effect and logical assertions which the execution of the task forces to be true.
- **Unimplemented tasks** have no information about how they are to be accomplished, either in the form of methods or in the form of primitive information such as logical assertions. Such tasks cannot be immediately executed; they must, instead, be adapted into either non-primitive tasks or primitive tasks.

**Methods:** Methods are behavioral elements: a description of a method encodes how that piece of computation works. A method description contains a state-transition machine which describes the operation of the method. The machine contains links to lower level tasks which are required to create the overall effect of the method. Thus tasks and methods are arranged in a hierarchy: tasks refer to methods which accomplish them and methods refer to tasks which are a part of them, all the way down to primitive tasks which have a direct specification of their effects.

**Knowledge:** Knowledge is the foundation on which tasks and methods are built. A description of a task or a method is inherently interconnected with the description of the knowledge that it manipulates. REM uses Loom [3,11] as its underlying mechanism for knowledge representation. Loom provides many of the basic

capabilities which are common to many knowledge representation formalisms, e.g., concepts, instances, relations, etc. The knowledge portion of TMKL begins with the basic Loom syntax and ontology and adds additional forms and terms for integrating this knowledge with information about the tasks and methods of an agent. Of particular value to REM is Loom's ability to represent knowledge about knowledge such as relations which describe properties of other relations. The adaptation of tasks and methods in REM is facilitated by abstract knowledge about the kinds of knowledge that these tasks and methods manipulate; the relation mapping algorithm in Section 4 provides an example of this sort of reasoning.

## 3 Illustrative Example

We have conducted experiments using REM on a variety of agents. A key element of these experiments involves the comparison of the combined effects of generative planning, reinforcement learning, and model-based reasoning with the effects of the separate approaches in isolation. One particularly noteworthy set of experiments involves the use of REM on ADDAM [6], a case-based agent which plans and executes (in simulation) the disassembly of physical devices. In this section we describe ADDAM and REM's use of it. The first subsection describes ADDAM itself, using REM's model of ADDAM. The second subsection then explains how REM uses this model to adapt ADDAM to assemble (rather than disassemble) devices. Finally, the last subsection describes the results of an experiment with REM and ADDAM involving the assembly problem.

### 3.1 The ADDAM Case-Based Disassembly Agent

Figure 1 shows some of the tasks and methods of ADDAM. Note that the TMKL model of ADDAM encoded in REM contains about twice as many tasks and methods as are shown here. In the interests of succinctness, we are only presenting the pieces of ADDAM which are directly relevant to this paper. The omitted elements occur in between and around the ones presented; they are largely focussed on supporting ADDAM's complex hierarchical structures for plans and devices.

The top level task of ADDAM is Disassemble. This task is implemented by ADDAM's process of planning and then executing disassembly. Planning in ADDAM involves taking an existing plan for disassembling a similar device and adapting it into a plan for the new device. ADDAM's planning is divided into two major portions: Make Plan Hierarchy and Map Dependencies. Make Plan Hierarchy involves constructing plan steps, e.g., screw Screw-2-1 into Board-2-1 and Board-2-2. The heart of this process is the creation of a node for the hierarchical plan structure and the addition of that node into that structure. Map Dependencies involves imposing ordering dependencies on steps in the plan, e.g., the two boards must be put into position before they can be screwed together. The heart of this process is the selection a potential ordering dependency and the

assertion of that dependency for that plan. Dependencies are much simpler than plan nodes; a dependency is just a binary relation over two nodes while a node involves an action type, some objects, and information about is position in the hierarchy. The relative simplicity of dependencies is reflected in the implementation of the primitive task which asserts them; this task is implemented by a simple logical assertion which says that the given dependency holds. In contrast, the tasks which make a plan node and add it to a plan are implemented by procedures.

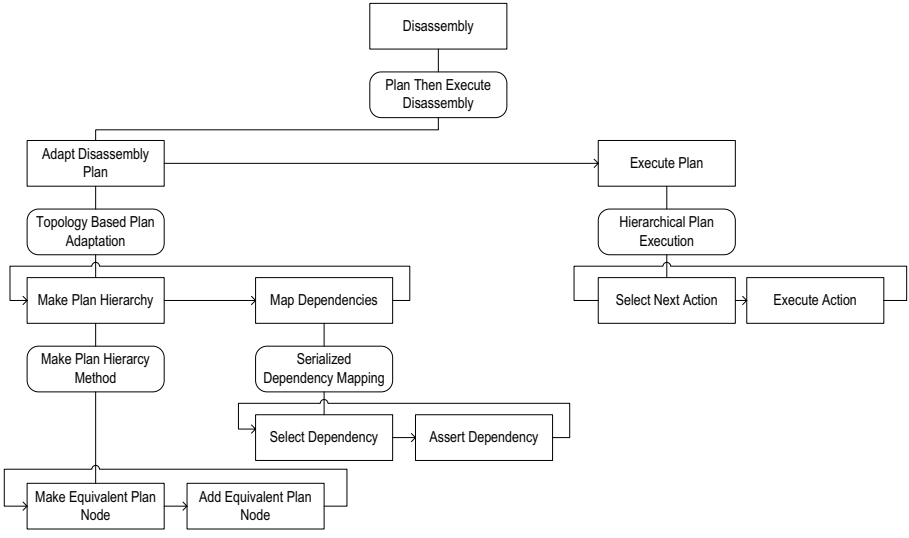


**Fig. 1.** A diagram of some of the tasks and methods of ADDAM. Rectangular boxes represent tasks, and rounded boxes represent methods.

Given the collection of plan steps and ordering dependencies which the ADDAM planning process produces, the ADDAM execution process is able to perform these actions in a simulated physical environment. Execution involves repeatedly selecting an action from the plan (obeying the ordering dependencies) and then executing that action. If the plan is correct, the ultimate result of the execution process is the complete disassembly of the simulated physical object.

### 3.2 Meta-case-Based Assembly with ADDAM in REM

ADDAM is able to perform disassembly on its own without any use of meta-cases. However, if ADDAM alone were given an assembly problem, it would be completely helpless. In contrast, when ADDAM is explicitly modeled within the REM reasoning shell, REM treats its model of ADDAM as a meta-case. When REM receives a request for assembly, it searches its meta-case memory for an implemented task which is similar to assembly. Because disassembly and

assembly are similar, REM retrieves ADDAM's disassembly task. At that point, REM can attempt to adapt that disassembly task to satisfy the new requirement: assembly.

REM is able to perform this adaptation using a combination of both model-based and non-symbolic techniques. In particular, REM applies model-based adaptation to ADDAM to provide an incomplete (underconstrained) assembly agent and then runs that agent repeatedly using reinforcement learning to produce a final, fully operational version of the assembly system. We have contrasted this approach with two other ways of addressing the same assembly problems; both of these alternative approaches are also performed by REM. The first alternative involves planning assembly through pure generative planning. This alternative is implemented by exporting the relevant facts and operations to Graphplan [2], a popular generative planning system. The second alternative involves performing assembly through pure reinforcement learning, i.e., beginning by simply attempting random actions on random objects and eventually learning which sorts of actions lead to successful assembly. This approach is implemented using the well-known Q-learning[1] algorithm [17].

When REM operating on ADDAM is asked to solve the assembly task, it is not given an implementation for that task. However, it is given the other portions of the TMKL model for a task: the inputs and outputs and the assertions which hold before and after the task executes. This information can be used to retrieve a similar task. In particular, the background knowledge in REM's model of ADDAM provides a relation, inverse-of, and an assertion that this relation holds between the states referred to in the output conditions of assembly and disassembly. This allows REM's meta-case retrieval mechanism to find the disassembly task.

REM has multiple strategies for adapting meta-cases. The one which is relevant to the assembly example is called relation mapping; it involves propagating a relation which holds between the desired task and the retrieved task through the implementation of the later to produce an implementation of the former. REM's relation mapping mechanism is a complex adaptation strategy involving changes to several different tasks within the overall structure of ADDAM. Furthermore, the process is not an entirely complete or reliable one; it involves suggesting a variety of modifications but some of these changes are only tentative and may conflict with each other. These incomplete modifications are resolved later in the reasoning process.

[1] Although Q-learning is a powerful and easy to use technique for decision making, it does have extensive competition, and there is undoubtedly some benefit to other decision making approaches. For example, TD($\lambda$) [15] is a reinforcement technique which is particularly well-suited to situations in which rewards are often delayed, which is typically the case in running a TMK model (since the reward comes at the end, when it is possible to observe whether the desired result of the specified task has been accomplished). Alternatively, one could envision abandoning reinforcement learning altogether and simply using a search process to make decisions. The benefits and drawbacks of using different techniques is an important area for future research.

Figure 2 presents the effects of relation mapping over ADDAM in the assembly example, again focusing on those elements which are relevant to this discussion. After the disassembly process is copied, there are three major changes made to the copied version of the model:

1. The Make Plan Hierarchy process is modified to adjust the type of actions produced. Because the primitive tasks which manipulate plan nodes are implemented by procedures, REM is not able to directly modify their implementation. Instead, it inserts a new mapping task in between the primitive tasks. The mapping task alters an action after it is created but before it is included in the plan. This newly constructed task asserts that the action type of the new node is the one mapped by the inverse-of relation to the old action type; for example, an unscrew action in an old disassembly plan would be mapped to a screw action in a new assembly plan.
2. The portion of the Map Dependencies process which asserts a dependency is modified. Because the primitive task for asserting dependencies is implemented as a simple logical assertion, it is possible to impose the inverse-of relation on top of that assertion. If one action was to occur before another action in the old disassembly plan then the related action in the new assembly occurs after the other related action (because inverse-of holds between the relations indicating before and after). For example, if an old disassembly plan requires that boards be unscrewed before they can be removed, the new assembly plan will require that they be placed before they can be screwed together.
3. The execution process is modified to adjust the type of actions executed. This adjustment is done by an inserted task which maps an action type to its inverse. For example, if a screw action is selected, an unscrew action is performed.

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

Fortunately, it is not necessary for REM's model-based adaptation technique to deductively prove that any particular combination of suggestions is consistent. Instead, REM can simply execute the adapted system with the particular decisions about which modifications to use left unspecified. In the example, REM

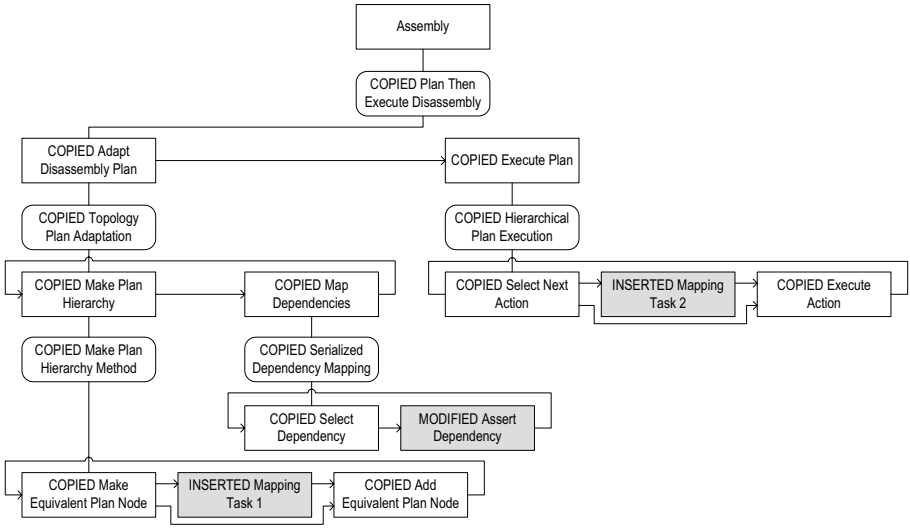


**Fig. 2.** Tasks and methods produced for the assembly process by adapting ADDAM. Tasks which were added or modified are highlighted in grey.

makes the two inserted mapping tasks optional, i.e., the state-transition machine for the modified methods has one transition which goes into the inserted task and one which goes around it. During execution, a decision making process selects among these two transitions; this decision making process uses Q-learning to resolve ambiguities in the model such as the optional mapping tasks. For the assembly problem, this decision making process develops a policy of including either of the inserted mapping tasks but not both.

Note that REM is using exactly the same decision making component here that it uses to perform this task by pure Q-learning; however, here this component is being used *only* to decide among the options which model-based adaptation left unspecified. In contrast, the pure Q-learning approach uses the decision maker 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 original problem (particularly if the original problem is, itself, complex); this fact is strongly reflected in the results.

### 3.3 Results

We have used REM on ADDAM for disassembly and assembly of a variety of devices such as cameras, computers, and furniture. One particularly interesting design which REM on ADDAM has been applied to is a nested roof design in which the placement of the upper level boards blocks access to the connections among the lower level boards. This design can be extended by adding more and more boards, thereby nesting the roof arbitrarily deep. This characteristic is

useful from an experimental perspective because it allows us to track the behavior of different approaches to assembly as the number of components increases.

Figure 3 shows the performance of REM on the roof assembly problem. Two of the lines indicate the performance of REM without access to ADDAM and the model thereof; one of these attempts uses Graphplan and the other uses Q-learning alone. The remaining line shows the performance of REM when it does use ADDAM, using the model to invert the system and then using reinforcement learning to fully specify the details of the inversion. The key observation about these results is that both Graphplan and Q-learning undergo an enormous explosion in the cost of execution (several orders of magnitude) with respect to the number of boards; in contrast, REM's model-based adaptation (with assistance from Q-learning) shows relatively steady performance. The reason for the steady performance is that much of the work done with this approach involves adapting the agent itself. The cost of the model adaptation process is completely unaffected by the complexity of the particular object (in this case, the roof) being assembled because it does not access that information in any way; i.e., it inverts the existing specialized disassembly agent to be a specialized assembly agent. The next part of the process *uses* that specialized assembly agent to perform the assembly of the given roof design; the cost of this part of the process is affected by the complexity of the roof design, but to a much smaller extent than generative planning or reinforcement learning techniques are.

## 4 Algorithms

The previous section contains a detailed example of the execution of REM for a specific problem. This section provides a more abstract view of the processes that are involved in that example. Table 1 shows a high-level overview of the main algorithm for REM. Note that this algorithm is essentially a case-based one: when there is a request for a task for which no method is available, a similar task is retrieved, the method for that task is adapted, the new method is stored for later reuse, and the new method is also verified through execution (and further adapted if needed).

REM has two inputs: main-task and initial-condition. In the earlier example, the main-task is assembly and the initial-condition describes the design of the device to be assembled (in this case, a roof) and the initial state of the world (that nothing is connected to anything else). If the task is unimplemented then a new method needs to be developed for it. In this situation, if there is some existing task which has a known implementation and is similar to the main task then we can try to adapt the implementation of the known task to address the new task (this process is described in more detail below). If no such existing task is available, then a method can be built from scratch, either by using a generative planner or by deferring the decisions about action selection until execution, thus reasoning purely by reinforcement learning. These three alternatives (adapting a method, building a method from scratch using generative planning,

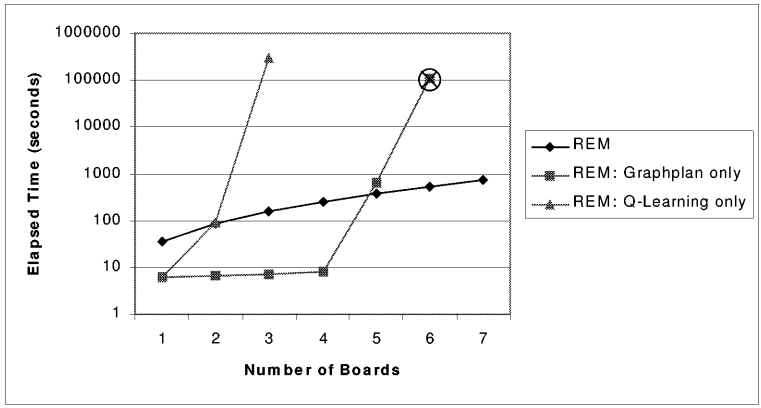


**Fig. 3.** Logarithmic scale graph of the relative performances of different techniques within REM on the roof assembly example for a varying number of boards. The Graphplan and Q-learning lines show the performance of these techniques used in isolation without any model information (the "X" through the last point on the Graphplan line indicates that the program, using Graphplan, ran up to that point and then terminated unsuccessfully, apparently due to memory management problems either with Graphplan or with REM's use of it). The other line shows the combined effect of REM's redesign of the model of ADDAM (which uses both model-based adaptation and Q-learning). Of particular note is the relatively steady performance of model-based adaptation, compared to the explosions in execution time from the other approaches.

**Table 1.** Main algorithm for REM

| Algorithm **do-task**(main-task, initial-condition) | |
|---|---|
| *Inputs:* | main-task: A task to be executed |
| | initial-condition: The input knowledge for the main-task |
| *Effects:* | The task has been executed. |

```
IF [main-task does not have a known implementation] THEN
   IF [there is a known-task which is similar to main-task] THEN
      [adapt the methods of that known-task to perform main-task]
   ELSE
      [build a method for main-task from scratch]

REPEAT
     [execute main-task under initial-condition]
     IF [execution has failed] THEN
        [modify the implementation of main-task]
UNTIL [execution has succeeded]
```

building a method from scratch using reinforcement learning) are contrasted in the experimental results presented in the previous section.

Once the main-task has an implementation, it can be executed in the context of the initial-condition. Decisions (such as method selection) which are left unresolved by the model are addressed through reinforcement learning during this execution process. When executing a method developed for pure reinforcement learning, all possible actions on all possible objects are available as separate methods for each step of the process; consequently, the decision making process is responsible for all of the control in this situation. In contrast, when REM's generative planning module (using Graphplan) has been used to build the method, a complete, ordered specification of exactly which actions to perform is available so that *no* decision making is needed during execution. Executing a method which was developed by model-based adaptation lies between the two extremes; many control decisions are specified by the adapted model, but a few are left open and must be resolved by the reinforcement learning process.

If execution is unsuccessful, REM can try to modify the implementation of the task and try again. If the task involves decisions which are being made through reinforcement learning, this modification can simply involve providing negative reinforcement to the last action taken by the agent (since it directly lead to failure). In other circumstances, REM is able to use model-based techniques to further adapt the implementation using the results of the execution and even user feedback (when available). No post-execution model-based adaptation is done in the ADDAM roof example, but we have conducted other experiments which do use these mechanisms. If additional modification is done, the REM process then starts over at the beginning, this time with an implemented task (so it can immediately start executing).

Recall that one of the steps described in the main algorithm of REM involves adapting the methods of a known task to perform main-task. This step is a complicated part of the process and there are many strategies which can be used for this step. Table 2 presents the algorithm for the particular adaptation strategy which is used in the ADDAM inversion example: relation mapping.

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

In the next step in the adaptation process, the system finds a relation which provides a mapping between the effects of main-task and the effects of known-task. In the experiment described in this paper, the relation which is used is inverse-of. This relation is selected by REM because (i) the task of disassembly has the intended effect that the object is disassembled, (ii) the task of assembly has the intended effect that the object is assembled, and (iii) the relation inverse-of holds between the assembled and disassembled world states. These three facts

**Table 2.** Algorithm for relation mapping, the adaptation strategy used in the assembly example.

| Algorithm **relation-mapping**(main-task, known-task) | |
|---|---|
| *Inputs:* | main-task: An unimplemented task |
| | known-task: An existing implemented task which is similar to main-task |
| *Effects:* | There is an implementation for main-task. |

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

are explicitly encoded in the TMK model of ADDAM. Note that there are other relationships besides inversion which could potentially be used by this algorithm or one similar to it. Some examples include specialization, generalization, composition, similarity, etc.; additional research is needed to fully explore the space of relations which can be used here and how those relations interact with the steps of this algorithm.

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

cause this change to take place, i.e., inverting the process of creating a disassembled state needs to be done implicitly by inverting internal information which leads to this state (such as plan node ordering information). Thus node-precedes and node-follows are included in the list of relevant manipulable relations while relations over world states are not.

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

## 5 Discussion

One obvious alternative to using REM in combination with a specialized task-specific case-based planner such as ADDAM is to simply use a general-purpose case-based planner. However, note that the assembly plans produced by REM and ADDAM bear virtually no superficial resemblance to the original disassembly plans upon which they were based: there is typically no overlap at all in the operators used (since all of the operators are inverted) and while the objects of those operators are similar, they are manipulated in the reverse order. Case-based reasoning is generally suited to the types of problems for which similar requirements lead to similar results; clearly the adaptation of disassembly plans into assembly plans does not meet that criterion. However, the processes by which the disassembly plans and assembly plans are produced are very similar (as evidenced by the relatively small differences between the models illustrated in Figures 1 and 2). Consequently, while this problem is ill-suited to traditional case-based reasoning, it is well-suited to meta-case-based reasoning. We claim that meta-case-based reasoning is also appropriate for many other problems for which similar requirements lead to radically different solutions but do so through relatively similar processes.

There are other case-based reasoning approaches which explicitly reason about process. For example, case-based adaptation [10] considers the reuse of adaptation processes within case-based reasoning. Case-based adaptation does not use complex models of reasoning, because it restricts its reasoning about processes to a single portion of case-based reasoning (adaptation) and assumes a single strategy (rule-based search). This limits the applicability of the approach to the (admittedly quite large) set of problems for which adaptation by rule-

based search can be effectively reused. It does provide an advantage over our approach in that it does not require a functional model and does not require any representation of the other portions of the case-based reasoning process. However, given that that case-based reasoning systems are designed and built by humans in the first place, information about the function and composition of these systems should be available to their builders (or a separate analyst who has access to documentation describing the architecture of the system) [1]. 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.

Another approach to using knowledge of process within case-based reasoning is derivational analogy [16]. Derivational analogy focuses not on abstract models of how reasoning can occur but rather on concrete traces of specific instances of reasoning. This can be an advantage in that these traces can be automatically generated and thus do not necessarily require any prior knowledge engineering. However, to the extent that the traces are automatically generated, derivational analogy is limited to kinds of problems which are already adequately solved by other approaches (typically generative planning). This can be effective if the agent first encounters relatively small examples which can be solved relatively efficiently using generative planning and then only encounters larger problems after it has built up some experience. However, if the environment provides a relatively complex example immediately, then derivational analogy is essentially helpless. In contrast, our meta-case-based reasoning approach is able to immediately address complex problems by leveraging the relative efficiency of some specialized case-based technique even on a distinct (but similar) task from the one for which that technique was designed.

Experiments with REM, such as the one presented in Section 3, show that meta-case-based reasoning using the combination of model-based adaptation and reinforcement learning is often computationally much less costly than reinforcement learning alone (and also less costly than generative planning). Note that reasoning about models does involve some processing overhead, and for very simple problems this overhead can exceed the benefits provided by model-based adaptation. However, if an agent is being built for a complex environment, its designer can expect that it will face complex problems, at least some of the time, and the enormous benefits provided by model-based adaptation for those problems should outweigh the additional costs for very simple problems. We, therefore, conclude that meta-case-based reasoning via a combination of model-based adaptation and reinforcement learning represents a reasonable and useful compromise among flexibility, knowledge requirements and computational cost.

## References

1. Gregory Abowd, Ashok K. Goel, Dean F. Jerding, Michael McCracken, Melody Moore, J. William Murdock, Colin Potts, Spencer Rugaber, and Linda Wills. MORALE – Mission oriented architectural legacy evolution. In *Proceedings International Conference on Software Maintenance 97*, Bari, Italy, 1997.

2. Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Aritificial Intelligence*, 90:281–300, 1997.
3. David Brill. Loom reference manual. *http://www.isi.edu/isd/LOOM/ documentation/manual/ quickguide.html*, December 1993. Accessed August 1999.
4. Michael Freed, Bruce Krulwich, Lawrence Birnbaum, and Gregg Collins. Reasoning about performance intentions. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pages 7–12, 1992.
5. Ashok K. Goel. A model-based approach to case adaptation. pages 143–148, August 1991.
6. Ashok K. Goel, Ethan Beisher, and David Rosen. Adaptive process planning. Poster Session of the 10th International Symposium on Methodologies for Intelligent Systems, 1997.
7. Ashok K. Goel, Sambasiva R. Bhatta, and Eleni Stroulia. Kritik: An early case-based design system. In M. L. Maher and P. Pu, editors, *Issues and Applications of Case-Based Reasoning to Design*. Lawrence Erlbaum Associates, 1997.
8. Ashok K. Goel and J. William Murdock. Meta-cases: Explaining case-based reasoning. In I. Smith and B. Faltings, editors, *Proceedings of the Third European Workshop on Case-Based Reasoning - EWCBR-96*, Lausanne, Switzerland, November 1996. Springer.
9. Todd Griffith and J. William Murdock. The role of reflection in scientific exploration. In *Proceedings of the Twentieth Annual Conference of the Cognitive Science Society*, 1998.
10. David B. Leake, Andrew Kinley, and David Wilson. Learning to improve case adaptation by intropsective reasoning and cbr. In *Proceedings of the First International Conference on Case-Based Reasoning - ICCBR-95*, Sesimbra, Portugal, 1995.
11. Robert MacGregor. Retrospective on Loom. *http://www.isi.edu/ isd/ LOOM/ papers/ macgregor/ Loom_Retrospective.html*, 1999. Accessed August 1999.
12. J. William Murdock and Ashok K. Goel. An adaptive meeting scheduling agent. In *Proceedings of the First Asia-Pacific Conference on Intelligent Agent Technology - IAT-99*, Hong Kong, 1999.
13. Eleni Stroulia and Ashok K. Goel. A model-based approach to blame assignment: Revising the reasoning steps of problem solvers. In *Proceedings of the National Conference on Artificial Intelligence - AAAI-96*, Portland, Oregon, August 1996.
14. Eleni Stroulia and Ashok K. Goel. Redesigning a problem-solver's operators to improve solution quality. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence - IJCAI-97*, pages 562–567, San Francisco, August 23–29 1997. Morgan Kaufmann Publishers.
15. Richard Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.
16. Manuela Veloso. PRODIGY / ANALOGY: Analogical reasoning in general problem solving. In *Topics in Case-Based Reasoning*, pages 33–50. Springer Verlag, 1994.
17. Christopher Watkins. *Learning from delayed rewards*. Ph.D. thesis, Cambridge University, Psychology Dept., Cambridge, UK, 1989.