Localizing Planning with Functional Process Models

J. William Murdock
Navy Center for Applied Research in A.I.
Naval Research Laboratory, Code 5515
4555 Overlook Avenue, SW
Washington, DC 20375-5337

Ashok K. Goel
Artificial Intelligence Laboratory
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

Abstract

In this paper we describe a compromise between generative planning and special-purpose software. Hierarchical functional models are used by an intelligent system to represent its own processes for both acting and reasoning. Since these models are custom-built for a specific set of situations, they can provide efficiency comparable to special-purpose software in those situations. Furthermore, the models can be automatically modified. In this way, the specialized power of the models can be leveraged even in situations for which they were not originally intended. When a model cannot address some or all of a problem, an off-the-shelf generative planning system is used to construct a new sequence of actions which can be added to the model. Thus portions of the process which were previously understood are addressed with the efficiency of a specialized reasoning process, and portions of the process which were previously unknown are addressed with the flexibility of generative planning. The REM reasoning shell provides both the language for encoding functional models of processes and the algorithms for executing and adapting these models.

Introduction

Generative planning is an extremely flexible approach to reasoning; when a planning system is given a new goal and it has an appropriate set of actions available, the planner can eventually find a combination of actions that accomplish the goal. However, the search for a set of actions that accomplish some goal can be extremely time consuming. In comparison, a specialized program that accomplishes a single specific effect (or some small set of effects) is likely to be much quicker; such a program does not need to search for a combination of actions to perform because its specialized code tells it exactly what to do and when to do it. Unfortunately, specialized programs are, by definition, extremely inflexible; they cannot be used for any purpose outside of their limited competency.

The trade-off between the flexibility of planning systems and the efficiency of specialized software suggests that we search for a mechanism that forms a compromise between the two. In particular, functional models can be used by an intelligent system to represent its own processes; these processes can include a combination of both computation and action. Because the models are specific to certain situations, they provide efficiency as special-purpose software does. Because the models can be automatically adapted, they can also be used in situations for which they were not originally intended. When a model cannot address some or all of a problem, generative planing can construct a new sequence of actions that can be added to the model.

The REM (Reflective Evolutionary Mind) reasoning shell provides capabilities for executing and adapting processes encoded in functional models. These models describe what the process does, what the portions of the process are, and how they work. When REM is asked to accomplish some effect that it already knows how to perform, it simply executes the known process. When REM is asked to accomplish some different effect, some changes need to be made. If some portions of a known process are relevant to the new desired effect, then they can be reused. If additional capabilities are also needed, then they can be added to the model. In this paper, we focus specifically on one of the techniques that REM has for adding new capabilities to a model: generative planning. A key benefit that generative planning in the context of a model provides over generative planning without a model is localization. Because the model already provides some portions of the process, generative planning is limited to portions which are not yet known. The experiments presented in this paper demonstrate that this localization can provide substantial performance improvements versus planning without prior knowledge of processes.

Models

Systems in REM are modeled using the Task-Method-Knowledge Language (TMKL). Processes in TMKL are divided into tasks, methods, and knowledge. A task is a unit of computation which produces a specified result. A task answers the question: *what* does this piece of computation do? A method is a unit of computation which produces a result in a specified manner. A method answers the question: *how* does this piece of computation work? Tasks encode functional information; the production of the specified result is the function of a computation. The knowledge portion of the model describes the different concepts and relations that tasks and methods in the model can use and affect as well as logical axioms and other inference mechanisms involving

those concepts and relations. Formally, a TMKL model consists of a tuple (T, M, K) in which T is a set of tasks, M is a set of methods, and K is a knowledge base.

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

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

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

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

A method in TMKL is a tuple (pr, ad, st) encoding a provided condition, additional results condition, and a state-transition machine. The two conditions encode incidental requirements and results of performing a task that are specific to that particular method of doing so. For example, a method for movement that involved driving a car would have a provided condition that a car be available and an additional results condition that the car has moved and has consumed some gasoline.

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

The representation of knowledge (K) in TMKL is done using using Loom (MacGregor 1999), an off-the-shelf knowledge representation (KR) framework. Loom provides not only all of the KR capabilities found in typical AI planning system (the ability to assert logical atoms, to query whether a logical expression holds in the current state, etc.) but also an enormous variety of more advanced features (logical axioms, truth maintenance, multiple inheritance, etc.). Planners that operate by reasoning forward from a start state to a goal state (Bacchus & Kabanza 1996; Nau *et al.* 1999) typically include some (but not all) of the advanced KR capabilities found in Loom, because reasoning forward provides constant access to a complete concrete state, making advanced KR feasible. REM also reasons in a forward direction. Furthermore, it does no backtracking because TMKL models can represent integrated planning *and* acting, and the latter cannot be reversed by simply considering an alternative. The lack of any need to return to an intermediate knowledge state makes it practical for REM to use such a powerful KR as Loom.

Models in TMKL resemble Hierarchical Task Networks (HTN's) in that both involve tasks that are decomposed by methods into partially ordered sets of subtasks. There are two key distinctions between TMKL and typical HTN formalisms (e.g., Erol, Hendler, & Nau 1994).

1) As noted above, TMKL models may include primitive tasks that are external actions that must be performed immediately in the real world (e.g., retrieving a file from a network, moving a robot, checking a sensor, etc.). While such tasks include conditions that specify their effects, those specifications can be incomplete and some effects may not be deterministic. This is a minor difference in the semantics of the formalism, but it has significant impact on the way that systems are defined. A typical HTN has numerous points at which alternative decisions can be made (e.g., multiple applicable methods for a task or multiple orderings of subtasks for a methods); HTN's tend to be relatively descriptive, i.e., they encode all the different ways that a task may be accomplished rather than prescribing a particular strategy for addressing a task. In contrast, TMKL models tend to be much more prescriptive; most branches or loops in a typical TMKL process have precise, mutually exclusive conditions describing what path to take in each possible state. TMKL does allow some decision points to be unspecified; these decisions are resolved by reinforcement learning. However, since a TMKL system cannot backtrack, and instead must start its reasoning over from the beginning whenever it reaches a failed state, acceptable performance is only possible when there are very few unspecified decision points.

2) TMKL encodes information in tasks and methods that is not directly needed for performing those tasks and methods but that is important for determining when a system has failed, for modifying a system to correct a failure, and for 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 implementa-

tion for the new task. In contrast, HTN's are primarily used in systems that assume a complete and correct HTN exists and that only attempt tasks encoded in the HTN. When automatically learning of HTN's is done (Ilghami *et al.* 2002), existing HTN's are not reused and extensive supervision is required.

Algorithms

The overall algorithm for the REM reasoning shell starts when a user supplies a description of a task and the values for the input parameters of that task. If the task is one for which there is already at least one method in the system, REM executes the task immediately. Otherwise, REM first uses an adaptation mechanism to build a method for the task. If REM knows another task with similar input and output parameters and given and makes conditions, it attempts to transfer existing method information from that task. Otherwise, it builds an entirely new method for the task from scratch, e.g., by generative planning. In either case, once adaptation is complete, REM is able to execute the task. After execution is done, if the makes condition is satisfied and the user has no objection to the results, then REM is done. Otherwise, REM performs additional adaptation, followed by more execution. The cycle of execution and adaptation continues until either the task is successfully addressed or REM has exhausted all available adaptation techniques. In this following subsections, the execution and adaptation algorithms in REM are described.

Execution

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

There are three additional issues regarding the execution algorithm: selection, trace generation, and failure monitoring. The issue of selection comes up when choosing a method for a non-primitive task and when choosing a transition within a method. Both methods and transitions have logical conditions which specify when they are applicable; if more than one option is available at any point in the process, then reinforcement learning, specifically Q-learning (Watkins & Dayan 1992), is used to choose among those options (Q-learning can be a very time consuming learning mechanism for large state-spaces, so it is important to limit models in REM to only a very small number of decision points that allow more than one option). When the desired effect for the main task is supplied by the user has been accomplished, positive reinforcement is provided to the system, allowing it to learn to choose options which tend to lead to success.

The issue of trace generation is addressed in REM by producing a record of what happened at each step of the process. The steps of the trace are encoded as TMKL knowledge structures in Loom, and are linked directly to the TMKL tasks, methods, states, transitions, etc. These traces are used if adaptation is required after execution; the traces provide a particular path through the model, and thus can guide and constrain the search for elements in the model which may need to be modified in response to the execution.

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

Model-Based Adaptation

One variety of adaptation which occurs in REM is *failure-driven model-based adaptation*. This approach begins by analyzing feedback from the user, if any, and adding additional failure annotations to the trace to indicate portions of the execution which contradict the feedback; no user-feedback is provided in the examples discussed later in this paper, but experiments in other domains have made extensive use of REM's ability to incorporate guidance from the user. After analying the feedback, the various failure annotations produced either during execution or during the aforementioned analysis are considered, one at a time. A variety of heuristics are used to order the consideration of annotations; these heuristics primarily focus on prioritizing the kinds of annotations for which more effective repair mechanisms are available. For each annotation considered, REM attempts to make some repair which could potentially prevent that failure from occuring. For example, if an annotation is encountered that indicates that there was no applicable method for a particular task in a particular situation, then REM attempts to construct a method for that task. In some situations, REM is able to make a repair via purely model-based reasoning. In other situations, additional reasoning mechanisms such as generative planning may be required to address a particular failure annotation.

REM can also build a method for a new task by modifying a method for a similar existing task to accomplish the new result. This process is referred to as *proactive model-based adaptation*. Because the task to be performed is new, REM cannot attempt to execute the task and then react to failures if necessary; instead it must perform adaptation before any execution can occur. Proactive model-based adaptation begins by retrieving a known task with a similar description to the given task. Then the differences in the descriptions (the input, output, given, and makes slots) of the two tasks are computed. Next a method for the retrieved task is copied over to the given task. That copy is then modified

by searching it for pieces (tasks and methods) which relate to the computed differences between the given and retrieved main tasks. In some situations, it can be unclear whether a particular change is appropriate; in those situations, REM sets up two alternatives (with and without that change) and allows the execution process to select among them using the Q-learning mechanism.

REM has a variety of algorithms for performing failure-driven and proactive adaption. Some of these algorithms are specific to one or the other application; however, the adaptation by generative planning algorithms (described below) can be applied in either case. Other REM adaptation mechanisms are described in similar detail in (Murdock 2001).

Adaptation by Generative Planning

The adaptation by generative planning algorithm creates a method for a task by using an external planning system to create a sequence of actions which accomplishes the indicated effect of the task. There are two situations in which this mechanism can be employed: (i) when a user specifies a new task to perform and that that task has no existing method, and (ii) when failure-driven model-based adaptation has identified a failure annotation which indicates that a method needs to be added to a task. In the latter case, the failure annotation indicates some task within an existing process that requires adaptation (because the annotation is linked to a step in the trace and the step in the trace is linked to the piece of the model that was executed at that step). Thus generative planning in that case is localized to a specific portion of an existing model-encoded reasoning process.

REM uses Graphplan[1] (Blum & Furst 1997) as its external planner. Table 1 presents the algorithm for the REM adaptation strategy that uses generative planning. The first few steps involve taking information in REM and translating it into a form which Graphplan can manipulate (facts and operators). The computation of the operators involves translating actions (those primitive tasks that affect external relations) into a form that can be processed by Graphplan; the given and makes conditions are translated into preconditions and postconditions for the Graphplan operator. If a primitive task involves Lisp code, that code is omitted from the operator. Note that not all TMKL actions may be able to be translated in to Graphplan's operator language. The results of a task are described in REM as an arbitrary Loom logical expression; such expressions can contain a wide variety of constructs which are not supported by Graphplan (e.g., disjunction, quantification, etc.). Furthermore, primitive tasks with procedures are not required to have their results stated at all; those primitive tasks whose results are omitted or cannot be translated into Graphplan's planning language are not included in the set of operators. The inability to translate all possible TMKL actions is an inevitable limitation of REM's planning mechanism: the planner can only reason about those actions that it can represent. The use of a different planning algorithm with a different language for representing operators could allow some operators which cannot be represented in Graphplan.

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

operators = [translate primitive-actions]
relevant-relations = [all relations in any slot of main-task or any of primitive-actions]
assertions = [translate all facts in knowledge-state involving any relevant-relations]
goal = [translate main-task:makes]
objects = [names of all objects which are directly referenced in facts or goal]
object-types = [all concepts for which the objects are instances]
plan = [**invoke planner** on operators, assertions, objects object-types, and goal]
new-method = NEW method
[subtasks for new-method] = [extract actions from plan]
[transitions for new-method] = [infer transitions from plan]
[add new-method to main-task:by-mmethod]
RETURN main-task

Table 1: Algorithm for adaptation by planning

After the operators are translated, the assertions, which constitute the starting condition for the planner, are extracted. Obtaining these assertions involves first finding the various relations that are relevant to the main task and the actions, i.e., those referred to in the logical expressions which describe the requirements and effects of those tasks. Next, the process finds all connections between values in the initial state involving those relevant relations. These assertions are also translated into a form that Graphplan can process. Then the goal (which appears in the makes slot of the main task) is translated. Finally, the objects and types in the domain are translated as well.

After all of the translation is done, the results are sent to two separate files which Graphplan requires as inputs: a "facts" file, which contains the objects, assertions, and goal, plus an "operators" file, which contains the operators. After the files are produced, Graphplan is executed and a plan is returned. The next few steps of the algorithm involve converting the plan into a method. First, a new method is produced. The subtasks of that method are simply the actions in the plan. A particularly challenging aspect of converting a plan into a TMKL method involves inferring transitions between the subtasks. Transitions are used both to guide

[1]The code for Graphplan was downloaded from http://www.cs.cmu.edu/afs/cs.cmu.edu/usr/avrim/Planning/Graphplan/. The Graphplan executable used in the work described in this dissertation was compiled with the following constant values: MAXMAXNODES was set to 32768 and NUMINTS was set to 1024.

the ordering of subtasks within a method and to indicate the bindings among the parameters of the subtasks. Building a method from a plan involves generating both of these types of information.

In principle, building the ordering can be very complex; for example, if a plan consists of two actions repeated several times, it might be reasonable to infer that the general process should contain a loop. It is not always clear, however, when to infer that a particular trend in ordering really does represent a significant pattern which should be represented in the method. The adaptation by planning mechanism in REM uses a very simple solution to the ordering problem: it puts the steps of the method in the same order as the steps of the plan with no branches or loops. It may be productive for future work on this topic to consider a broader range of alternatives for ordering steps in a method which was constructed from a plan.

REM's mechanism for constructing bindings for the parameters of subtasks is somewhat more sophisticated. That mechanism infers bindings by first assigning parameters to the input values and then propagating the parameter-value relationships through the plan; this process is essentially a form of Explanation-Based Generalization (Mitchell, Keller, & Kedar-Cabelli 1986). The method produced 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 bindings among the parameters (in the transitions) are a form of explanation in that they tie the values to the parameters and thus explain how the effects of the actions in the plan (expressed as specific values) accomplish the overall goal of the main task (typically expressed in terms of the task's parameters). Once the transitions for the method are completely specified, the method is ready to be used. The specification of the main task is revised to indicate that the newly constructed method is one which can accomplish that task. At that point, adaptation is done.

Illustrative Example

In this section we describe the use of REM on an illustrative planning domain. The domain used is inspired by the Depot domain in the Third International Planning Competition (2002); however, the version presented here 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, can only be placed in one of three locations (the warehouse, a loading pallet, and the truck); that part of the problem is isomorphic to the Tower of Hanoi problem, a traditional illustrative example for AI and planning. In addition, we have eliminated the route planning and unloading aspects of the problem. In our version, there is only one possible place to drive the truck, and once the truck arrives, the driver only needs to provide documentation for the shipment 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.

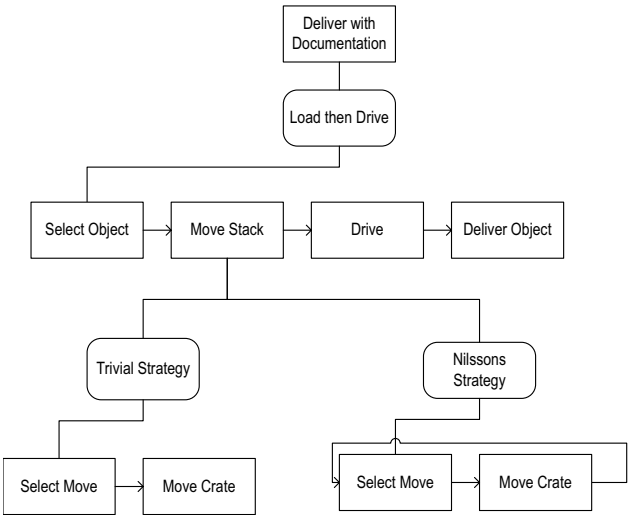

Figure 1: Tasks and methods of the example system.

Figure 1 shows the tasks and methods of the example system in REM which addresses this problem. The overall task, Deliver with Documentation, is decomposed into four subtasks: Select Object, Move Stack, Drive, and Deliver Object. The first task outputs an object to deliver; specifically, it always outputs the documentation for the shipment. The second task moves the crates from the start location to the end location, using the intermediate location if necessary. It has two separate methods: Trivial Strategy and Nilsson's Strategy. The former has a provided condition which must hold to be executed: that there be exactly one crate. That strategy simply moves the crate to the truck. Nilsson's Strategy, on the other hand, has no provided condition, i.e., it can be used for any number of crates. It uses a relatively efficient iterative technique created for the Tower of Hanoi problem; specifically, it is a generalization of the Tower of Hanoi strategy in (Nilsson 1998, errata, p. 4).

The two methods for the Move Stack task are redundant; the behavior of Nilsson's Strategy when there is only one crate is identical to the behavior of Trivial Strategy. Both methods are included to allow example problems which involve one or the other. For example, some of the results presented in the next section involve 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 failure-driven model-based adaptation using generative planning, which eventually leads to a correct solution.

Results

In this section we present some experimental results involving REM. The first subsection presents results from the ex-

ample problem described in the previous section. The results which we have obtained for this problem concretely illustrate the benefits of localization of planning using REM. The second subsection then briefly describes results from a much more significant domain: physical device disassembly and assembly. These results show similar benefits to those found in the example problem. More details on the device disassembly experiments, plus details of experiments involving web browsing, meeting scheduling, and other domains are all found in (Murdock 2001).

Example Problem Results

The main task in the TMKL model for the example problem is Deliver with Documentation. Figure 2 presents the total time used in performing this task in a variety of different situations with a varying number of crates. The data points show results for a single run, but the qualitative differences in the data are unchanged across multiple runs. The first data series shows the performance of the full system; because this system has methods which work for any number of crates, no adaptation of any sort is needed. The second series shows the performance of the ablated system in which Nilsson's Strategy has been removed. The ablated system is able to perform correctly when there is only one crate, but it fails when it encounters more than one crate. When it does fail, generative planning repair is used to build an additional method for Move Stack. The third series shows performance for the generative planning strategy when there is no use of an existing model at all (i.e., generative planning is used for the entire main task).

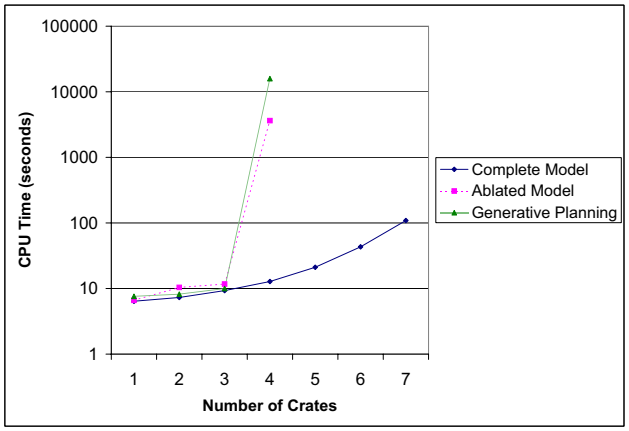

Figure 2: Logarithmic scale graph of the relative time taken by REM for the Deliver with Documentation task using different mechanisms with different knowledge conditions.

There are several key insights which can be observed from Figure 2. One insight is the fact that the curves that involve planning undergo a sudden, dramatic jump in cost even at a relatively small problem size. In contrast, the technique that uses only the specialized process encoded in the model shows relatively steady performance The most interesting aspect of this data is the relative performances of the two series that involve generative planning (the second and third). The performance data for these two systems are similar to each other but not identical. For one crate, the ablated system is slightly faster than pure generative planning. The ablated system does not need to do any adaptation for one crate because the method that it does have can handle one crate. The pure generative planning process is also very quick for the one crate case, but not quite as fast as the system which already knows how to solve the problem.

For two and three crates, the performance of the ablated system is slightly worse than the performance using pure generative planning. Whenever the number of crates is greater than one, the process for the ablated system involves running the system, encountering a failure, selecting a particular localization of the failure (as identified by trace annotations), planning at that localization, and then executing the corrected system. Figure 3 shows the ablated system after adaptation in the two crate example. In contrast, the pure generative planning process involves just planning and executing. Figure 4 shows the pure generative planning system after adaptation in the two crate example. The process for the ablated system involves a more complex model, plus an additional execution step, plus localization. These factors clearly impose some additional cost over the pure generative planning system (although it does appear from the data that this cost is fairly small). Of course, the ablated system does have one advantage: in the planning step, the ablated system only needs to form a plan for moving the stack of crates. In contrast, pure generative planning needs to plan the entire delivery process process (involving not only moving the crates but also driving and delivering the documentation).

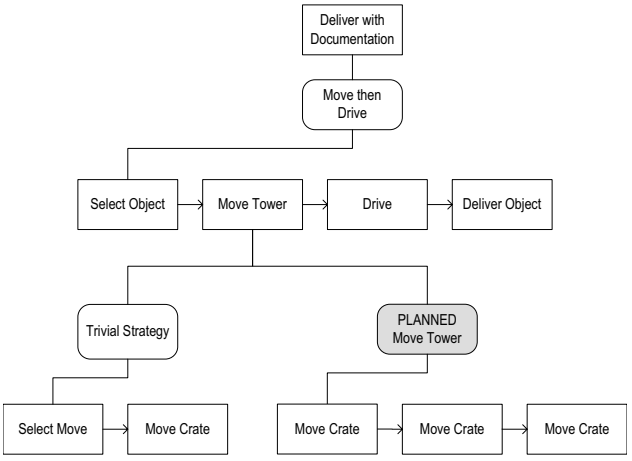

Figure 3: The ablated system after being executed with two crates and then adapted using generative planning repair. The newly constructed method is highlighted in grey. The model produced for more than two crates is similar but includes more Move Crate actions.

In the two and three crate examples, the advantage that the ablated system has in addressing a simpler planning problem appears to be outweighed by the additional model-based reasoning that it performs. However, in the four crate example the balance is shifted. Both processes show a dramatic jump in cost at the fourth crate. Note, however, that the cost for the

ablated system is lower by a noticeably wider margin than the advantage that the other approach had at two and three crates (even on the logarithmic scale graph). Specifically, the ablated system took approximately an hour while the pure generative planning system took about four and a half hours. Recall that that the only additional steps that need to be included in the plan for the pure generative planning system are driving and delivering and that only one possible destination was available for driving. In the smaller examples, the entire process involving moving crates, driving, and delivering is performed by the pure generative planning system in under ten seconds. However, in the four crate example, planning these extra steps adds approximately three and a half hours to the overall process, more than quadrupling the total cost. The example problem is easily decomposed into moving the crates, driving, and delivering; however, the pure generative planning mechanism has no prior knowledge of that decomposition and must be constantly considering the issues involved in those extra steps throughout dealing with the crate moving issues. In contrast, the ablated system already has the problem decomposed via its TMKL model. It is thus able to use the Graphplan only on the portion of the process that it does not already know how to solve. This experiment has shown that decomposition and localization via TMKL can provide a substantial advantage for planning.

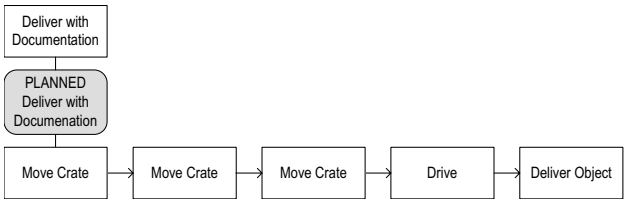

Figure 4: The example problem after being executed with two crates and no pre-existing model; pure generative planning is used to produce the adapted system. The newly constructed method is highlighted in grey. The result for more than two crates is similar but includes more Move Crate actions.

Physical Device Disassembly Results

One system which has been encoded in REM is ADDAM (Goel, Beisher, & Rosen 1997), a physical device disassembly agent. ADDAM was not created as part of this research, and the original ADDAM system did not contain any representation of itself and thus did not have any ability to modify its reasoning processes. The combination of REM and ADDAM has been tested on a variety of devices, including cameras, computers, furniture, etc. A nested roof design is one example which has undergone particularly extensive experimentation in the course of this research. A key experimental feature of that roof design is the variability in its number of components; the design involves screwing together a set of boards together at particular angels, and can be easily extended and retracted by adding and subtracting boards. This variability is useful for experimentation because it is possible to see how different reasoning techniques compare on the same problem at different scales.

A particular task which has been of interest in experiments with REM and ADDAM has been assembly. ADDAM, being a disassembly system, has no method for assembly. Consequently, in order to address this task it must first perform some adaptation. REM is able to build a method for assembly by modifying ADDAM's method for disassembly via proactive model-based adaptation. However, if the method for disassembly is ablated from the system, then REM must construct a method from scratch using generative planning (or another adaptation mechanism).

The performance of REM on roof assembly problem is presented in Figure 5. The first data series involves the performance of REM when it uses ADDAM via proactive model-based adaptation. The other data series involves the performance of REM without access to ADDAM's disassembly method, using generative planning.[2] The key observation about these results is that planning undergoes an enormous explosion in the cost of execution (several orders of magnitude) with respect to the number of boards; in contrast, REM's proactive model-based adaptation shows relatively steady performance.

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

Note again the resemblance between Figure 5 and Figure 2. The results for the assembly task provide meaningful validation for the properties suggested in the crate delivery results, i.e., that the reuse of process information from a functional model can provide enormous performance benefits over planning in isolation for complex problems.

Discussion

As noted earlier, TMKL models are very similar to HTN's. Lotem, Nau and Hendler (1999) have combined HTN planning with the Graphplan algorithm. However, that work focused on using Graphplan to order subtasks within a known method, i.e., to speed up planning with an existing HTN. In contrast, REM uses Graphplan for learning how to address new tasks and for learning new methods to address existing tasks

Like REM, PRODIGY (Veloso *et al.* 1995) employs machine learning techniques to enhance planning. PRODIGY's

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

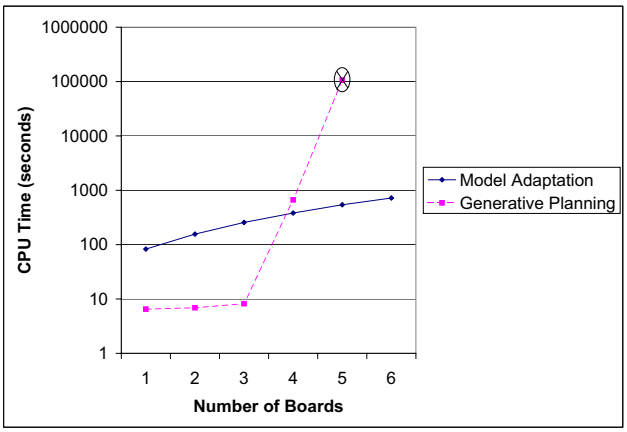

Figure 5: Logarithmic scale graph of the relative performances of model-based adaptation and generative planning techniques within REM on the roof assembly example for a varying number of boards. The "X" through the last point on the Graphplan line indicates abnormal termination (see text).

enhancements to planning are used for a wide range of effects, such as improving efficiency via derivational analogy (Veloso 1994) and improving plan quality via explanation-based learning (Pérez & Carbonell 1994). However, these techniques assume that there is a single underlying reasoning process (the generative planning algorithm) and focus on fine tuning that process. This is very effective for problems which are already well-suited to generative planning, making them even more well-suited to planning as the planner becomes tuned to the domain. However, it does not address problems which are ill-suited to planning to begin with. If a problem is prohibitively costly to solve at all by generative planning, then there is no opportunity for a system learning about the planning process to have even a single example with which to try to improve the process. In contrast, systems in REM can start with a model that encodes an efficient process for a problem or part of a problem. This model can enable REM to effectively reason and act in domains for which pure generative planning is infeasible. REM's adaptation mechanisms can then allow parts of the model to be reused for new (but related) problems.

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

Like case-based planning, model-based adaptation exploits similarity between problems. However, while case-based planners demand that similar problems lead to similar sequences (or hierarchies (Muñoz-Avila *et al.* 1999)) of actions, model-based adaptation instead demands that similar problems lead to similar reasoning processes (even if those reasoning processes lead to very different actions). Consider, for example, the disassembly to assembly adaptation example. The assembly plans in these examples bear virtually no superficial resemblance to the original disassembly plans upon which they were based: there is typically no overlap at all in the operators used (since assembly involves actions like placing and fastening while disassembly involves actions like removing and unfastening). While the objects of the operators are similar, they are generally manipulated in a very different order. However, the processes by which the disassembly plans and assembly plans are produced are very similar. Consequently, while this problem is ill-suited to traditional case-based reasoning, it is well-suited to model-based adaptation.

Functional models of processes are a substantial requirement that REM has and that traditional generative planning systems do not. However, given that that 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 to a separate analyst who has access to documentation describing the architecture of the system (Abowd *et al.* 1997). Thus while our approach does demand significant extra knowledge, that requirement is evidently often attainable, at least for well-organized and well-understood systems. Our results have shown that this additional knowledge can be used to provide substantial performance benefits.

References

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

Bacchus, F., and Kabanza, F. 1996. Using temporal logic to control search in a forward chaining planner. In Ghallab, M., and Milani, A., eds., *New Directions in AI Planning*. IOS Press (Amsterdam). 141–156.

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

Erol, K.; Hendler, J.; and Nau, D. 1994. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth*

National Conference on Artificial Intelligence - AAAI-94. Seattle, WA: AAAI Press.

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

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

Ilghami, O.; Nau, D.; Muñoz-Avila, H.; and Aha, D. W. 2002. CaMeL: Learning methods for HTN planning. In *Proceedings of the Sixth International Conference on AI Planning and Scheduling - AIPS-02*. Toulouse, France: AAAI Press.

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

Lotem, A.; Nau, D.; and Hendler, J. 1999. Using planning graphs for solving htn problems. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence - AAAI-99*, 534–540. Orlando, FL: AAAI Press.

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

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

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

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

Murdock, J. W. 2001. *Self-Improvement through Self-Understanding: Model-Based Reflection for Agent Adaptation*. Ph.D. thesis, Georgia Institute of Technology, College of Computing, Atlanta, GA. http://thesis.murdocks.org.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In Thomas, D., ed., *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence - IJCAI-99*, 968–975. Morgan Kaufmann Publishers.

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

Pérez, M. A., and Carbonell, J. G. 1994. Control knowledge to improve plan quality. In *Proceedings of the Second International Conference on AI Planning Systems*.

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

Veloso, M. 1994. PRODIGY / ANALOGY: Analogical reasoning in general problem solving. In *Topics in Case-Based Reasoning*. Springer Verlag. 33–50.

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