Model-Based Reflection for Agent Evolution

J. William Murdock
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

Ph.D. Defense of Research
December 20, 2000
Technical Report GIT-CC-00-34

Thesis Committee:
Ashok Goel (Advisor), Alex Kirlik, Sven Koenig,
Colin Potts, Ashwin Ram, Spencer Rugaber

Abstract

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

In particular, my research examines the use of reflection (an agent's knowledge and reasoning about itself) to accomplish evolution (incremental adaptation of an agent's capabilities). I have developed a language called TMKL (Task-Method-Knowledge Language) that enables modeling of an agent's composition and functioning. A TMKL model of an agent explicitly represents the tasks the agent addresses, the methods it applies, and the knowledge it uses. TMKL models are hierarchical, i.e., they represents tasks, methods and knowledge at multiple levels of abstraction. I have also developed a reasoning shell called REM (Reflective Evolutionary Mind) which provides support for the execution and evolution of agents represented in TMKL.

REM employs a variety of strategies for evolving TMKL agents. Some of these strategies are purely model-based: knowledge of composition and functioning encoded in TMKL directly enables adaptation. REM also employs two traditional artificial intelligence and machine learning techniques: generative planning and reinforcement learning. The combination of model-based adaptation, generative planning, and reinforcement learning constitutes a mechanism for reflective agent evolution which is capable of addressing a variety of problems to which none of these individual approaches alone is suited. My research demonstrates the computational feasibility of this mechanism using experiments involving a variety of intelligent software agents in a variety of domains.

1 Issues

Many environments require that a computing system respond effectively to a very wide range of situations. The intelligent agent research paradigm focuses on environments of this sort. In particular, intelligent agents often interact heavily with both human users and other computational systems and thus need to be able to address the broad range of input information that these sources can provide. Because intelligent agents typically deal with a wide variety of inputs, they are generally designed to be "flexible" in the sense that they have a wide variety of behaviors built into them. However, this sense of the word "flexible" is very limited; they may have many capabilities encoded in them, but a sufficiently dynamic environment will always demand additional capabilities which the agent's designer did not anticipate or did not have the resources to provide. Environments of this sort demand an agent which can evolve to meet the demands of novel situations.

Much traditional research in Artificial Intelligence has addressed the issue of generalized strategies for providing flexible behavior. Two such approaches are generative planning and reinforcement learning. Generative planning [Tate et al., 1990] is an extreme form of deliberative reasoning; it makes no use of environmental cues or experience but rather addresses problems by constructing an appropriate sequence of actions using information about the requirements and effects of those actions. In contrast, reinforcement learning [Kaelbling et al., 1996] is an extreme form of situated reasoning; it makes no effort to predict the consequences of actions but rather addresses problems by acting and observing the effects that different actions have. Both of these approaches allow a reasoner to address an arbitrary goal, the former by constructing a plan which accomplishes the goal and the latter by simply acting and recognizing when the goal is met (and, in the process, learning to distinguish between effective actions and ineffective actions). However, both of these approaches have serious limitations regarding efficiency and knowledge requirements which severely restrict their applicability to the environments in which intelligent agents are typically needed.

One obvious way that the designer of an intelligent agent can avoid the limitations of these general reasoning approaches but still obtain some of the power and flexibility that they provide is to combine techniques; i.e., the designer can build customized software components to address pieces of the problem that are hard for general approaches and then just invoke the general approaches for those parts of the problem which those approaches can address. For example, one could construct a World-Wide-Web agent which used custom built routines to elicit a request from the user and then used generative planning to obtain data from the Web and then used reinforcement learning to select particular data items to present to the user. Such an agent would be very challenging and time-consuming to build and test, but, if done well, could potentially be very powerful. An even more appealing possibility would be to create a general-purpose reasoning shell which provides an agent designer with access to features such as reinforcement learning and generative planning. This could reduce the effort required to implement agents of this sort but would not address the (very challenging) problem of how to divide a large task into pieces which can be addressed by general approaches. Furthermore, an agent built in such a shell would only be able to reason flexibly for those pieces of its process which were explicitly instructed to use a flexible strategy. The remaining pieces of the agent and the overall architecture of the agent itself would be as rigid and inflexible as any other special-purpose intelligent agent.

The research presented here provides an alternative account of how a general-purpose reasoning shell can combine the flexibility of generative planning and reinforcement learning with the power of custom-built intelligent agent software. Instead of requiring an agent designer to explicitly divide a particular problem into pieces to be addressed by different approaches, this work only requires that the agent designer provide the shell with a single specialized implementation of the agent and a formalized description of how that agent works (i.e., a *model* of the agent). Whenever that agent encounters demands and resources that exactly match those that it was built for, no general planning or learning is required and it is able to simply operate as it is. However, when novel situations arise, the shell is able to intervene and provides the mechanisms for the agent to *evolve*. These mechanisms include not only specialized techniques which use the models of the agents to make specific modifications but also generative planning and reinforcement learning capabilities.

1.1 Generative Planning for Intelligent Agents

The primary problem with generative planning in agent domains is that it often takes an unacceptable amount of time to begin acting. Modern generative planners [Kautz & Selman, 1996, Blum & Furst, 1997, Weld et al., 1998] work much faster than planning systems of early eras. However, even with these planners, combining large numbers of very primitive operators is generally very expensive. Hierarchical planners [Sacerdoti, 1974, Tate, 1977, Knoblock, 1994] address this issue to some extent in that they allow reasoning at higher levels, and thus permit the agent to plan over a small number of operators (and then elaborate those operators). However, planning can be prohibitively expensive, even with a small number of operators, particularly in complex environments where many subgoals conflict with each other.

Furthermore, the languages used to specify planning operators are often inadequate to describe complex agent behavior (or can only do so using overwhelmingly cumbersome formalization). Planning systems typically specify the requirements and results of their use in a highly restricted language. Some planners only allow simple lists of atomic facts for preconditions, postcondition additions, and postcondition deletions. Most, at least allow operators to be parameterized (note, however, that some implement this by internally enumerating all possible parameterizations of each planning operator and then plan over this set; this is yet another potential source of intractability). Some planning systems deal with universal quantification [Penberthy & Weld, 1992] over a specific type of item (e.g., an operator which clears a hard-drive would have a delete list which specified that for all files, that file is no longer on the hard-drive). However, the item types used in these systems tend to be extremely simple, i.e., they generally do not even support subclassing, much less advanced features such as multiple inheritance, implicit classification, etc. Additionally, relatively few planning systems allow logical forms such as disjunction, implication, etc. Much work in the area of knowledge representation has suggested that some or all of these features are crucial for providing succinct, useful descriptions of complex domains, e.g., [Lenat & Guha, 1990, Genesreth & Fikes, 1991, MacGregor, 1999].

Of course, it is, in principle, possible to build a planner that addresses all of these issues. One could provide an operator language which specifies preconditions and postconditions of an operation in (for example) first-order predicate calculus. Unfortunately, experience (and common sense) has typically shown that the less restrictive an operator specification language is, the more time-consuming planning becomes. Given the severe tractability problem that planners have even for highly restrictive languages, it is clear that planning over a very powerful operator language is infeasible for all but the smallest of problems.

In addition, to these concerns, explicitly specifying all of the possible actions in a domain can require enormous knowledge engineering, even with an appropriate formalism for doing so. Common experience with declarative programming languages (such as Prolog) has shown that many problems require as much effort or more (often much more) to describe logically as they do to write an algorithm to solve. Many agent domains are sufficiently complex that explicitly specifying all possible actions in the domain is prohibitively difficult, even with a relatively expressive operator language and especially with a very restrictive language such as the ones that relatively fast planning systems use.

1.2 Reinforcement Learning for Intelligent Agents

Reinforcement learning also has serious limitations. While such an agent can begin acting immediately, these actions are not guided by any advance reasoning and thus can often be unproductive or even catastrophic. If the agent knows enough about the effects of actions to try them out in a simulated environment, it can "plan" by performing simulated situated action with reinforcement learning until an effective action policy is developed. This problem avoids many of the representational problems that exist for planning operators; i.e., the specification of the effects of an action that is being "planned" over by simulated reinforcement learning can easily include complex logical forms since the simulator only needs to assert these effects one at a time rather than reason about complex combinations of these effects. Unfortunately, this form of planning is often even more time consuming then traditional generative planning. Finding an effective action policy through exploration and numerical learning takes an enormous number of iterations, even for a moderately sized state space. Furthermore, many problems are not easily formalized as involving a parsimonious set of discrete states. Many problems demand a space of states that is exponential in the size of the problem being addressed. Given the extreme cost of reinforcement learning relative to the state-space size, the existence of exponentially many states poses an overwhelming obstacle except for extremely small problems.

Another limitation of reinforcement learning is that it requires information about how beneficial specific results of specific actions are (i.e., a reward function). Of course, an arbitrary goal can be transformed into a reward function by simply declaring that actions which cause the goal to be achieved provide a large reward while all other actions provide no reward. Reinforcement learning is capable of producing an optimal action policy for reward functions of this sort. However, doing so typically requires many, many more iterations than learning a policy in a comparable domain for which a more detailed reward function is available. For reinforcement learning to be a cost-effective approach to solving a specified goal, it is generally necessary to know how the results of individual actions contribute to the accomplishment of the goal; i.e., to reward not only actions that complete the goal but also actions which bring the system closer to the goal. However, for many domains (including most agent domains) there are clearly defined effects that a system is required to accomplish but little or no quantitative information about how much any particular action contributes to accomplishing that effect. For example, if a World-Wide-Web agent accesses a web page which may or may not have useful information, there is no immediate feedback about how useful that action is; such feedback only comes at the end of the task (if at all) when the user is presented with the information and is given an opportunity to use it.

2 Model-Based Reflective Adaptation

Given the extreme limitations of general-purpose reasoning paradigms, it is not surprising that the vast majority of software systems are built with a fixed set of inflexible behaviors. There is a thriving software industry based almost exclusively on special-purpose programs written by humans. The existence of this industry is clear evidence that people are capable of developing a wide variety of large and complex systems for solving specific computational problems. A major drawback of hand-built custom systems, of course, is that they are not flexible; i.e., because they were designed for a particular purpose, they are generally ill-suited to any other purpose. Generative planning and reinforcement learning avoid this drawback (since they can address an arbitrary goal) but suffer from the limitations listed above. Of course, some computational problems are insurmountably difficult or completely impossible, and thus cannot be handled even by custom-made software systems. However, common experience with the software industry has shown that specialized software can address a broad and interesting class of problems, including many large, complex problems which are far beyond the scope of generative planning or reinforcement learning.

One alternative to using general-purpose mechanisms is to start with some specialized systems and allow those systems to *adapt* themselves when some unexpected demand arises. Adaptation is the process of making a modification in response to the demands of an environment. A specialized software system may not need to adapt itself at all if the requests made of it and the resources available to it are exactly those for which it was designed. When these conditions do not hold, however, adaptation is generally necessary. Obviously an agent has a substantial advantage in making a modification to itself if it can *reflect* on knowledge about itself. Reflection is an approach to reasoning which focuses on the analysis and manipulation of knowledge about one's self. Thus any technique by which an agent modifies itself in response to its environment through the application of self-knowledge can be described as *reflective adaptation*.

One form of reflective adaptation involves the application of a library of adaptation strategies, using explicit *models* which describe the design of the agents being adapted. A model is an explicit representation of a phenomenon which supports inferences about both static and dynamic properties of that phenomenon and its elements. There are many different ways in which an agent can be modeled; one family of agent models which has been shown to provide useful information is the TMK (Task-Method-Knowledge) approach.[1] TMK models provide information about both the function of systems and their elements (i.e., the tasks that they address) and the behavior of those systems and elements (i.e., the methods that they use), using explicit representations of the information that these elements process (i.e., the knowledge that they apply). This approach obviously adds a fairly substantial knowledge

[1]The TMK approach has a complex history involving its use for a wide variety of purposes. Major antecedents of TMK include Structure-Behavior-Function (SBF) models physical devices [Goel, 1989, Goel et al., 1997b] and the theory of Generic Tasks [Chandrasekaran, 1988]. Models embodying the TMK approach have been referred to as SBF models [Stroulia & Goel, 1995], SBF-TMK models [Stroulia & Sorenson, 1998], and TMK models [Goel & Murdock, 1997, Griffith & Murdock, 1998, Murdock, 1998]

requirement; i.e., that models of this sort be available. However, given that these systems are designed and built by humans in the first place, this information should be available to the builder of the system (or a separate analyst who has access to documentation describing the architecture of the system) [Abowd et al., 1997]. Thus while the knowledge requirement here is significant, it is evidently often attainable. One aspect of the research presented here has been the development of a new formalism for TMK models called TMKL (for TMK Language). TMKL fits within the general framework of TMK, but provides more power and flexibility than previously existing formalizations of TMK.

3 Evolutionary Reasoning Shells

Reasoning over models of tasks, methods, and knowledge is particularly effective at identifying aspects of an existing system which require change and is also effective at making some sorts of changes. Existing work shows that this approach is effective at fixing subtle flaws in an existing system [Stroulia & Goel, 1995, Stroulia & Sorenson, 1998]. My dissertation research has included work on a system called SIRRINE which uses knowledge structures of this sort to enable the addition of new computational capabilities to a system. SIRRINE is a reasoning shell which executes agents encoded as TMK models, and, as necessary, analyzes the execution traces of the agents to identify elements to be modified and then modifies those elements using a collection of specialized adaptation strategies. SIRRINE provides further evidence for the effectiveness of TMK models in supporting localization of potential sources of change [Murdock & Goel, 1999b]. SIRRINE further demonstrates that localization of this sort can be used to make changes that alter the overall capabilities or constraints of a system [Murdock & Goel, 1999a]. However, the nature of the changes that SIRRINE makes is very limited. These limitations provide the motivation for another system which I have also worked on in my dissertation research: REM. Like SIRRINE, REM addresses the task of adding new capabilities to an existing agent. However, REM provides an approach to this problem which allows much more dramatic additions.

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

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

4 Illustrative Example

I have conducted experiments using REM and SIRRINE 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 [Goel et al., 1997a], a case-based system which plans and executes (in simulation) the disassembly of physical devices. I have used the combination of REM and ADDAM to address the task of assembly of devices; i.e., REM inverts ADDAM's specialized disassembly planning and execution method to produce a new method which assembles devices. REM's process of inversion uses model-based reflective adaptation 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. I 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 [Blum & Furst, 1997], 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 algorithm.

One of the experiments involving REM and ADDAM involves the assembly of a hypothetical layered roof design involving of a variable number of boards. The design is very simple, consisting only of boards and screw. However, the configuration of the roof is such that the placement of each new board obstructs the ability to screw together the previous boards so the assembly must be constructed in a single precise order, i.e., place two boards, screw them together, and then repeatedly place a board and screw it to the previous board until all boards are placed. When REM is asked to solve this problem it is provided with the following:

1. A description of the required task, i.e., assembling the roof, including a description of the topology of the roof itself.

2. A set of primitive assembly and disassembly actions (e.g., placing, removing, screwing, unscrewing, etc.) defined in TMKL. This TMKL description includes both the requirements and effects of these actions as well as some additional knowledge about them (e.g., that unscrewing is the inverse of screwing).

3. ADDAM, a specialized case-based disassembly planning system which contains a relevant library of disassembly plans (which involve some of the actions described above).

4. A TMKL model of ADDAM, including explicit references to its primitive actions (see item 2, above).

5. A statement that the required task (assembly) is the inverse of the disassembly task which ADDAM addresses.

Note that items 1 and 2 above directly provide the knowledge requirements for generative planning: the task encodes a starting state and a desired end state, while the primitive actions are operators. Furthermore, it is also possible to solve this problem using just 1 and 2 through situated action with reinforcement learning: the task describes a desired state, and the agent can simply pick actions at random and observe whether the desired state is met (and reinforce the corresponding decisions as needed). Indeed, REM is able to solve this particular problem using either of those approaches when only given knowledge items 1 and 2, above.

When REM is also given items 3, 4, and 5, it has a third strategy available to it: model-based reflection. In particular, item 5 (the statement that the required task is the inverse of disassembly) suggests that REM's inversion adaptation strategy is appropriate to solving the problem. The process of inversion is a complex one 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 modification but some of these changes are only tentative and may conflict with each other. The transition from disassembly planning to assembly planning in ADDAM requires not only that disassembly operators be inverted into assembly operators (e.g., unscrew is inverted into screw) but also that ordering constraints in the plans be inverted (e.g., if a piece must be unscrewed and then removed in the disassembly plan, it must be placed and then screwed in the assembly plan).

One of the changes suggested by REM is inverting the subtask of ADDAM's case-based planning process which adds specific actions to a plan (so that it adds the inverse operation to the plan). Another change suggested by REM involves the inversion of the planning subtask which asserts ordering constraints. Another change suggested by REM is inverting the subtask of ADDAM's plan execution which selects the next operation to perform (so that it produces the inverse of the next operation of the plan). Obviously the first and third suggestions conflict with each other; if the system inverts the operators when they are produced *and* when they are used, then the result will involve executing the original operators. 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 operations twice. However, the level of detail required to deduce these facts is not in this model (and REM's inversion strategy makes no attempt to do so). Even if the information were there, it would be in the form of logical assertions about the requirements and results of all of the intervening tasks (which are 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 assertions, 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 with each other. Instead, REM can simply execute the modified system with the particular decisions about which modifications to use left unspecified. These decisions can then be made during execution; if the correct combination of decisions is made, the assembly of the device will be successful. Thus to complete the adaptation of ADDAM, REM simply needs to use a decision making technique which begins without any information and, through experience with the success or failure of actions, converges upon an effective policy. Of course, this is precisely what reinforcement learning is for. Thus in this example, REM runs the partially specified assembly module and uses reinforcement learning to incrementally specify the correct configuration of inverted task. Note that REM is using the same Q-learning component here that it uses to perform this task by purely situated action; however, here this component is being used *only* to decide among the alternative configurations which model-base adaptation left unspecified. In contrast, the purely situated approach uses the Q-learning 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 reflection 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.

Figure 1 shows the performance of REM on roof assembly problem. Two of the lines indicate the performance of REM without access to ADDAM and the model thereof; these attempts use Graphplan and situated action (with Q-learning), respectively. 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 correct 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 reflection (with assistance from Q-learning) shows extremely steady performance. The reason for the relatively 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 planner to be a specialized assembly planner. The next part of the process *uses* that specialized assembly planner to perform the assembly of the given roof design; the cost of this part of the process is (obviously) affected by the complexity of the roof design, but to a much smaller extent than generative planning or reinforcement learning techniques are.

One important note about this experiment is that for each problem size, each strategy is only used once (i.e., REM is executed using that strategy until it generates a correct solution and then terminates). This evaluation is particularly unfair to the Q-learning module because when it is used for a single execution, it gets no benefits from learning (and is really just performing actions at random). Graphplan and model-based reflection, in contrast, both do some reasoning in advance, and thus are able to behave strategically, even for a single execution. Additional experiments (to be described in detail in my dissertation) have looked at the performance of these strategies on repetitions of the same problem. These results show

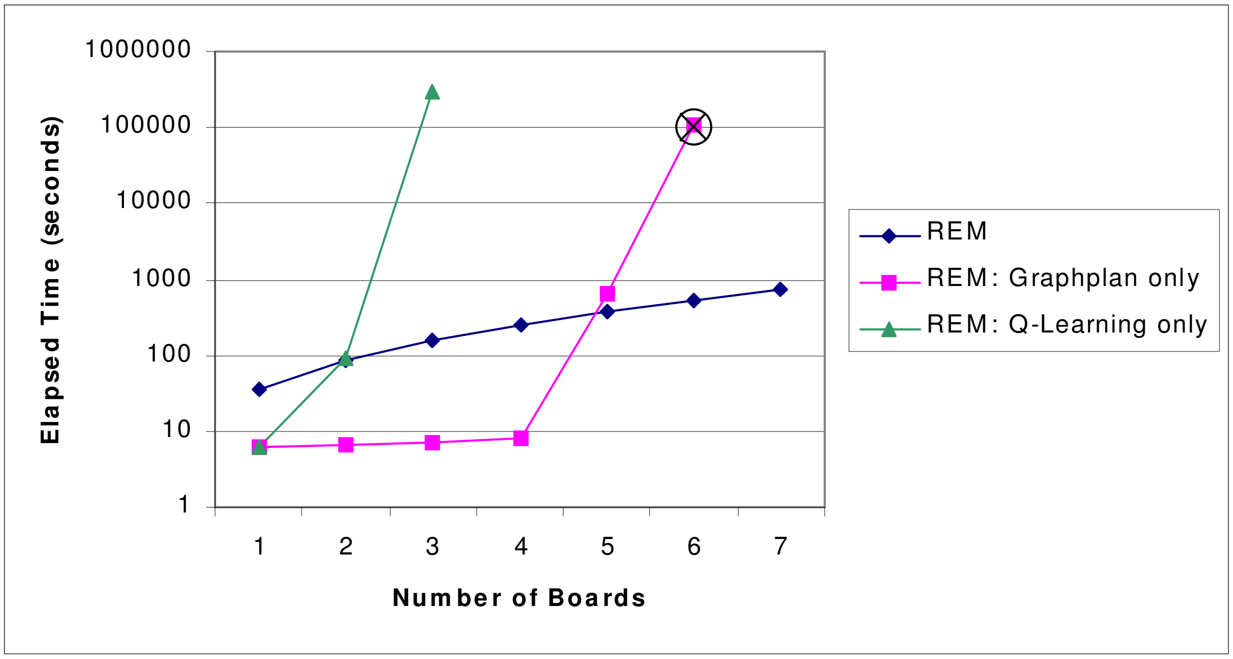

Figure 1: Logarithmic scale graph of the relative performances of different techniques within REM on the roof assembly example for a varying number of boards. The Graphplan and Q-learning lines show the performance of these techniques used in isolation without any model information (the "X" through the last point on the Graphplan line indicates that the program, using Graphplan, ran up to that point and then terminated unsuccessefully, 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 reflective evolution using the model of ADDAM (which uses both model-based reflection and Q-learning). Of particular note is the relatively steady performance of reflective evolution, compared to the explosions in execution time from the partial approaches.

that there are benefits to reusing learned information on additional problems (so Q-learning, in particular, is not quite as bad as this first set of results suggest), but generally suggest the same overall conclusions; i.e., that the overall effect of REM as a whole scales to complex problems more efficiently than generative planning or reinforcement learning alone do.

Figure 2 shows the relative performance of these different approaches in a related experiment. The same reasoning techniques are used in this experiment but the design of the roof to be assembled is slightly different: specifically, the placement of new boards does *not* obstruct the ability to screw together previous boards. These roof designs contain the same number of components and connections that the roof designs in the previous problem do. However, planning the assembly of these roofs using generative planning much easier than the ones in the previous experiment because the goals (having all the boards put in place and screwed together) do not conflict with each other. The results of these experiments demonstrate the fact that REM using only Graphplan is able to outperform the model-based reflection approach; i.e., because the problem itself is fundamentally easy, the additional cost of using and transforming a model of ADDAM outweighs any benefits that the specialized assembly planner provides. This illustrates an important point about model-based reflection: i.e., that it is ideally suited to problems of moderate to great complexity.

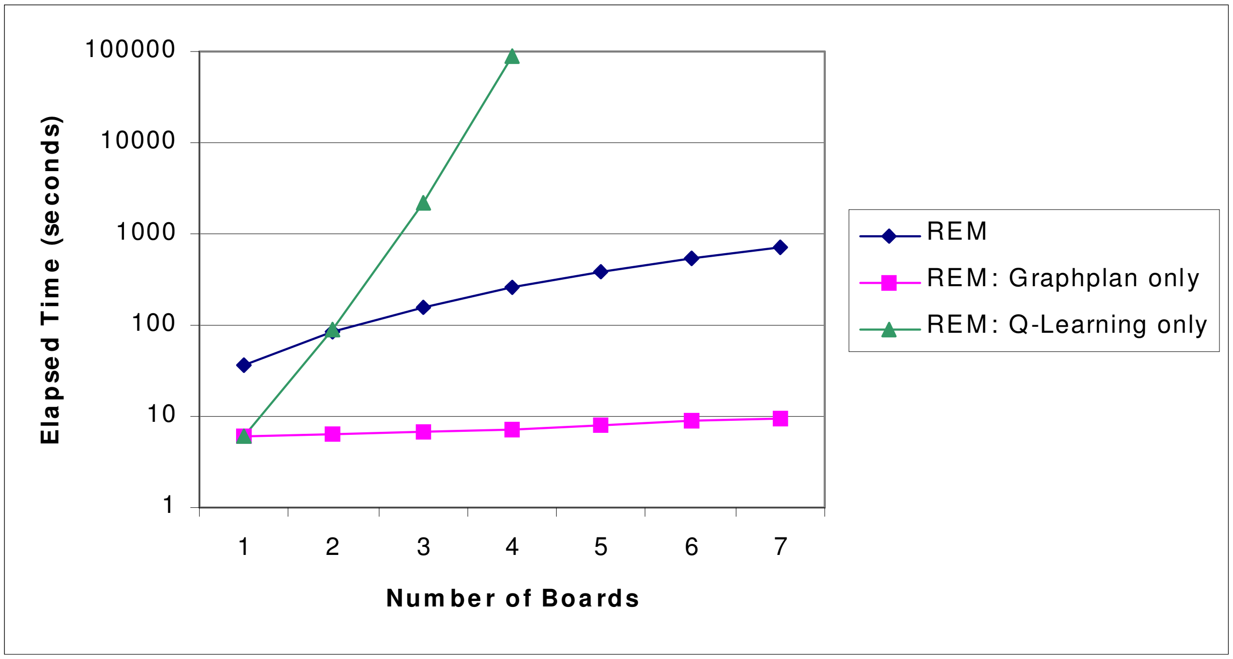

Figure 2: Logarithmic scale graph of the relative performances of different techniques within REM on a modified roof assembly example in which there are no conflicting goals. REM's reflective evolution still performs effectively, but does not outperform Graphplan which also scales well for problems of this sort.

The results of these and other experiments have shown that the combination of model-based inversion and reinforcement learning on ADDAM provides tractable performance even for devices which cannot be handled effectively by the alternative approaches. The pure reinforcement learning approach is overwhelmingly expensive for all but the most trivial devices. The Graphplan approach is extremely rapid for relatively simple devices and even scales fairly well to *some* devices which contain substantially more components. However, there are other, very similar devices which involve more complex relationships among the components for which Graphplan is completely overwhelmed. The inverted ADDAM system is able to handle devices of this sort much more quickly than Graphplan is.

5 Evolutionary Architecture

One significant feature of REM which is lacking in many reasoning shells based on Task-Method-Knowledge models (including SIRRINE) is that the capabilities of the shell, itself, is encoded in a TMK model. One of key advantages of this fact is that the model of the shell constitutes a specification of the overall design and theory of the shell; i.e., a description of the TMK model of REM *is* an abstract architecture for evolutionary agents. REM is one implementation of this architecture (obviously); SIRRINE also implements pieces of this architecture, but in a less principled manner. The description of the evolutionary architecture below is based on the TMK model of REM but focuses on general principles which may be more generally applicable. It begins with a description of the knowledge found in REM. Next, it describes the high-level tasks (which use and manipulate that knowledge). Finally, it describes the methods (which implement those tasks).

5.1 Evolutionary Reasoning Knowledge

The specific implementation of the TMK modeling approach encoded in REM is called TMKL.[2] This language constitutes the bulk of the Knowledge portion of REM's model of itself; i.e., the definition of TMKL constitutes most of the "K" portion of the TMKL model of REM. TMKL breaks down into three major portions: tasks, methods, and knowledge.

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 its input and output slots which identify the kinds of information the task uses and produces as well as its given and makes slots which identify propositions which are required to be true before and after the task is performed. 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; REM is able to do more direct and powerful manipulation of the later variety of tasks (since it can reason about logical assertions

[2]The relationship between TMK and TMKL can be a bit confusing. One can view this relationship as similar to the relationship between the concepts "skyscraper" and "the Empire State Building" in that the former is an abstract class of artifacts characterized by similarities in design and purpose while the latter is a specific instance of this class. The details presented in this section focus on TMKL.

explicitly), but the former is also provided for the sake of completeness (since some primitive tasks are not easily represented as logical assertions).

- **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 evolved 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 in TMKL is described by provided slot which specifies facts which must be true for the method to operate and an additional-results slot which specifies consequences of the operation of the method. Note that these slots are not intended to specify the overall function of the method (that information is encoded in the description of the task which the method addresses); rather these slots are intended to specify incidental requirements and results which are specific to a particular way of accomplishing the overall effect. For example the task of robot navigation might have a given state in which a robot is in one location and a makes state in which the robot is in another. One method for this task, case-based navigation, might a provided condition that a similar case is in memory, and an additional-result condition that a new case has also been stored in memory. Other methods might have provided clauses which involve availability or a map or of environmental sensors, etc.

In addition to the provided and additional-results slots, a method description also contains a state-transition machine which describes the operation of the method. The this 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 [Brill, 1993, MacGregor, 1999] as its underlying mechanism for knowledge representation. LOOM provides many of the basic capabilities which are common to the knowledge portions of other TMK formalisms (such as the TMK models in SIRRINE) including concepts, instances, relations, and propositions. In addition, LOOM adds a number of additional features which substantially add to the power of knowledge representation in REM and TMKL including implicit classification, universal and existential quantifiers, etc.

5.2 Evolutionary Reasoning Tasks

The primary task of REM is called solve-problem:

- **TASK** solve-problem
  - *input:* problem-state main-task
  - *output:* result-state result-trace

- *makes:* result-state is consistent with the makes condition of main-task
- *methods:* execute-then-evolve evolve-then-execute

This is the general task of solving an arbitrary problem: given a state of knowledge and a task, accomplish that task. For example, if REM were used on ADDAM to disassemble a given camera, the solve-problem task would be invoked with a main task of disassembly and a problem state which specifies that the device to disassemble is the camera. Of course, the execution of solve-problem will not always be successful (since REM, like any computing system, cannot solve every possible problem). If the task given to REM has a makes slot which describes what must be true for the task to have succeeded, then REM will be able to recognize whether the solving of the problem was successful or not.

One major subtask of solve-problem is execute. This task attempts to address a problem by simply using the existing implementation for the task; this invokes a TMKL interpreter which allows systems represented in TMKL to run. When the TMKL interpreter is invoked for a non-primitive task, it first selects a method for that task, and then it begins at the start state of that method and moves through the state-transition machine until it reaches a terminal state. Each state refers to a task to be invoked: the interpreter executes that task just as it does the main task for the agent, so if the a task referred to by a state in a method is non-primitive, the interpreter will select and execute a method for it. At the lowest level of the task-method hierarchy are the primitive tasks; the interpreter is able to directly accomplish the effect of these tasks. Most of the behavior of the interpreter is fairly straightforward and fully specified by the TMKL being executed. However, there are a couple of places for decisions to be made. For example, there may be more than one applicable method for a given task; decisions of this sort are made by a separate decision making module which uses reinforcement learning (specifically Q-learning) to develop an appropriate policy.

Another major subtask of solve-problem is evolve. The purpose of evolve is to make a modification to a task; this can occur before attempting to execute the task (which is essential if the task is unimplemented) or after attempting the task (at which time a trace is available, providing substantial benefits for evolution). REM has several methods for the evolve task; these are described in the following section.

5.3 Evolutionary Reasoning Methods

The solve-problem task has two methods: execute-then-evolve and evolve-then-execute. The former approach involves first attempting the task using the existing implementation and then only making adjustments to the task if necessary, while the latter attempts to evolve the task first and only after that tries to execute this task.

- **METHOD** execute-then-evolve
  - *provided:* main-task is not unimplemented
  - *subtasks:* execute elicit-feedback evolve solve-problem

The provided clause of this method states that it cannot be used for unimplemented tasks; if the task is not implemented (i.e., it has no methods and is not a primitive task) then it cannot be executed. The subtasks of a method are organized by a collection of states and transitions which specify the constraints on the order in which they operate. The execute-then-evolve method runs its four subtasks in sequential order; in addition, there are other transitions which cause the method to terminate if the execution is successful or to retry an execution using reinforcement learning, as needed. Three of the subtasks of this method are described in the previous section: (execute, evolve, and solve-problem). The remaining subtask, elicit-feedback, is a relatively simple one which obtains post-execution feedback from the user; for example, a user of a web browsing agent may specify something about a file which should have been accessed or a viewing program which should have been used. Note that the final subtask of execute-then-evolve is the main task for REM, solve-problem, i.e., after running and evolving the given task, REM recursively invokes the solve-problem to try the modified task again (and, if necessary, evolve it again).

The evolve-then-execute method can be attempted at any time; it has no provided clause because evolution can be attempted before execution whether or not the task being considered is implemented.

- **METHOD** evolve-then-execute
  - *subtasks:* evolve solve-problem

The method consists of only two subtasks: evolve and solve-problem. Note that the combination of these two methods (execute-then-evolve and evolve-then-execute) does allow for a task to be first evolved and then executed and then evolved and executed again: when the solve-problem is first invoked, the evolve-then-execute method can then be chosen, which invokes the solve-problem task again, for which the execute-then-evolve method can be chosen.

There are four methods for the evolve task: proactive-model-evolution, failure-driven-model-evolution, planning-evolution, and situator-evolution. The first two of these methods involve model-based adaptation. In contrast, the other two of these methods involve the use of relatively generic reasoning techniques which do not make extensive use of TMK model information.

The planning-evolution technique exports the primitive tasks of the agent to Graphplan and uses that planner to develop a plan to accomplish the effects of the given task under the given problem state; it then generalizes this plan into a method (primarily by variablizing specific instance values and tracing the flow of values to infer which variables in each step should correspond to each other). The method is expected to be appropriate to the given problem state (since it is based on a plan for that state); it is thus executed using the TMKL interpreter. Furthermore, the plan is stored so that it may be reused (and, if necessary, further evolved) in addressing other problems in the future.

The situator-evolution method is a particular simple technique: it simply creates a method which just executes actions without any *a priori* strategy and develops an effective policy for selecting actions by observing their consequences (i.e., situated reinforcement learning). Specifically, this approach creates a method with one subtask which it runs repeatedly until the desired result of the task is met. This subtask has a set of new methods each of which executes a specific primitive action for a specific combination of inputs. One such method is provided for each possible combination of action and inputs and no guidance is provided in the model for selecting among these methods (other than the known prerequisites of the primitive actions themselves). When the main task is then executed, the interpreter invokes its decision-making module, which uses Q-learning to develop a policy. Since all possible action / input combinations are available for selection, Q-learning provides all of the strategy and control for tasks evolved by situator-evolution.

The proactive-model-evolution involves taking an unimplemented task and providing a method for it by adapting a method used for that task. The example problem in Section 4 demonstrates the operation of proactive-model-evolution; in the data provided there, the performance of proactive evolution of the TMKL model of ADDAM is contrasted with methods developed by the situator-evolution and planning-evolution techniques.

The failure-driven-model-evolution also uses models to adapt a task. This method has a provided condition which states that a trace of the behavior of the main task under the given state be available. This requires that execution of the task have already taken place. Consequently, failure-driven-model-evolution is specifically appropriate as a method for the evolution part of the execute-then-evolve top-level method but not for the evolve-then-execute method.

The two strategies for model evolution represent the primary usage of TMK models in this evolutionary architecture. These two methods are inherently very complex and could potentially be addressed by a wide variety of subtasks (which could, in turn, invoke a wide variety of lower level methods); the details of these methods which have been developed in SIRRINE and REM have been shown to be broad and powerful enough to address a substantial variety of problems, but appear to have only scratched the surface of the wide variety of conceivable techniques for model-based adaptation. The development of additional model-based adaptation strategies seems like a particularly promising direction for future research.

6 Applicability and Limitations

The primary task of REM, solve-problem, is an extremely general one: given an arbitrary task and an arbitrary world-state, accomplish that task. Obviously, REM and SIRRINE cannot be successful for all possible tasks and states; in fact, it will not always be possible to determine whether this approach will succeed for a certain task. However, there are some relatively clear indications of problems for which this theory of reflective agent evolution is not particularly well suited. **Major limitations of this theory include:**

Extremely novel problems. If the task to be accomplished bears very little resemblance to any known tasks either in the form of the task or in the nature of the processing which can be used to accomplish that task, then reflection on models of existing tasks is unlikely to provide an effective means of accomplishing those tasks. In these situations, a tool like REM is forced to rely exclusively on first-principles reasoning techniques, which are severely limited in the kinds of problems they can solve (as noted earlier). The boundary for how novel a problem that model-based reflection can solve is primarily defined by the quantity and power of the adaptation strategies available; this is discussed in more detail in the list of limitations of SIRRINE and REM.

Extremely easy problems. Model-based reflection often involves a fair amount of overhead. For example, solving a new task using a model of a related task requires that the entire model be transformed before the task can be accomplished. In environments in which all (or nearly all) problems are known to be extremely simple and easily solved by known general-purpose techniques, the use of model-based reflection adds an unnecessary extra cost. In contrast, however, environments in which a mixture of simple and complex problems occur may well benefit from model-based reflection since the moderate extra cost incurred for very simple problems is likely to be outweighed by the potentially enormous savings for more challenging problems (for which the general-purpose techniques can be completely intractable).

Poorly understood systems. For reflective agent evolution to adapt an existing method for some task, that task and method (and the associated knowledge) must be explicitly represented. It is expected that this information is available about systems that are well understood (either by the system's designer or by a later analyst). However, when no such understanding is available, it is not possible to meet the knowledge requirements for reflective evolution. Fortunately, this limitation is mitigated by two facts: (i) because TMK models do not require that all of the requirements and effects be specified for every task, reflection over these models degrades gracefully when only partial information is available, and (ii) there has been substantial research in the area of program understanding, including work that explicitly considers the analysis of programs in the language of tasks and methods, e.g., [Abowd et al., 1997].

Poorly organized systems. Some systems are not accurately represented as relatively small collections of parsimonious tasks and methods. The most extreme possible variant of this issue is a large program which consists exclusively of a sequence of commands which have absolutely no relationship to each other regarding their intended effects. If such a program were encoded in TMKL, it could only be represented as a single high level task with a single method which consists of an enormous sequence of extremely miniscule primitive tasks. Such a model would likely be enormous and yet would contain no significant information which isn't directly represented in the code, itself; it would add extensive overhead but not provide any substantial leverage for making modifications to the system. Fortunately, common experience with software shows that this extreme is not found in actual software systems; functions, modules, blocks, etc. do generally demonstrate at least some degree of internal consistency regarding what these pieces are intended to do. However, many systems also display some degree of conceptual incoherence, e.g., the interleaving of pieces of code which serve distinct objectives [Rugaber et al., 1995]. Extreme incoherence of a system design greatly limits the usefulness of reflection over that design.

In addition to these fundamental limitations of the reflective evolution approach, there are also some more specific limitations of SIRRINE and REM which are not necessarily inherent to the overall theory. These limitations present clear opportunities for future research (since they describe capabilities that a successor to these systems could potentially have). **Major limitations of SIRRINE and REM include:**

Limited quantity of adaptation strategies. The model-based reflection techniques depend on a collection of specialized strategies to perform specific transformations of TMK models. REM has a larger and more powerful collection of adaptation strategies than SIRRINE does, but there is certainly the potential to add many more. Since the breadth of coverage provided by this approach is largely defined by the range of adaptation strategies, this issue provides the most dramatic potential for future enhancement of REM, SIRRINE, or a similar system.

Choice of Graphplan vs. other generative planners Graphplan has been shown to perform substantially better than many of its competitors [Blum & Furst, 1997] and is, consequently, a very popular planning system. The choice of Graphplan versus some other known planner has not been the focus of extensive study during this project because the emphasis has been on using model-based reflection to reduce the size of planning problems to a level at which they can be easily addressed, even by an expensive generative planner (so the issue of which expensive generative planner is of secondary importance). However, there are other planning systems which could provide significant benefits. One disadvantage that Graphplan has in its use in REM is that TMKL has a much more powerful language for describing primitive tasks than Graphplan has for describing operators; consequently, there are some primitives which can be represented in Graphplan only partially or not at all. There are many planning systems which would support a greater subset of TMKL, e.g., SGP [Weld et al., 1998], typically at the cost of even greater computation time. It is also possible that some planning systems could make more effective use of additional knowledge that a TMKL model might encode. There is evidence that forward chaining is a superior approach to planning when extensive domain knowledge is available [Bacchus & Kabanza, 1996]; it is not intuitively obvious whether the domain knowledge typically available in TMKL models is appropriate to gain these benefits. Also, since tasks and methods in TMKL are organized hierarchically, it is possible that task and method information could be provided to a hierarchical planner, e.g., [Sacerdoti, 1974, Tate, 1977, Knoblock, 1994, Nau et al., 1999]; precisely how to make such a translation is a topic which would probably require extensive additional research. In principle, one could connect REM to a variety of different planners and choose among them based on circumstances; of course, this idea raises the question of whether the selection process can be made efficient enough that it's cost doesn't outweigh the associated benefits.

Choice of Q-learning vs. other decision making techniques. Q-learning is a powerful and easy to use technique for decision making. However, it does have extensive competition; as with planning, the focus of the study of REM and Q-learning has involved using model-based reflection to accomplish enough of the problem that the size of the problems left to Q-learning are extremely small. However, there is undoubtedly some benefit to other decision making approaches. For example, TD(λ) [Sutton, 1988] 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 doing so is an important area for future research.

Non-functional requirements. Tasks in TMKL are described by their functional requirements (i.e., what needs to be true of the knowledge and world states before and after execution occurs). However, there are many other sorts of requirements that a user might have. The most obvious sort of non-functional requirement that a user might request is processing speed; i.e., the user may want a certain effect to be accomplished as quickly as possible, or before a certain time limit. To some extent, demands of this sort can be reformulated as functional requirements; for example, a task to assemble a device within a certain time could have functional specification which says that at the end of execution, the device is assembled *and* no more than some fixed amount of time has elapsed. If individual tasks had information about how they affect elapsed time then SIRRINE or REM could potentially solve a problem of this sort. However, this is an extremely limited view of reasoning about non-functional requirements. One problem with this approach is that it ignores the cost of the reasoning shell's own analysis, e.g., if REM were given a task to accomplish some goal and have only two hours elapsed, and it tried to do this by combining two subtasks each of which cause an hour to elapse, then the goal still would not be accomplished because the total time of execution would be two hours *plus* the time that REM used building and monitoring these tasks. Another problem with the approach is that both the representation and the processing used to accomplish arbitrary logical assertions may be generally ill-suited to the specific task of satisfying processing time requirements. There is extensive work on using reflection to affect processing speed [Russell & Wefald, 1991, Goodwin, 1994]. It is possible that reflective knowledge and reasoning of this sort could be combined with the functional reflection performed by SIRRINE and REM; this seems like an interesting topic for future research.

7 Results and Conclusions

The results of this research support the following claims:

- Model-based adaptation can provide tractable flexibility for an important range of problems. 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 relatively simple.

- For some other problems, model-based adaptation may only be able to provide a partial solution. This is commonly the case when there is a relevant existing strategy but the functional differences between the new problem and the old problem are more complex.

- Partial solutions provided by model-based adaptation require additional reasoning to complete. Completing these solutions can provide relatively small and simple challenges to general purpose reasoning techniques such as reinforcement learning and thus may be tractably completed by these techniques even when the complete problems themselves cannot be.

- Consequently, the combination of model-based adaptation, generative planning, and reinforcement learning can provide effective performance over a range of problem sizes for which both reinforcement learning alone and generative planning alone are shown to be thoroughly intractable and for which model-based adaptation alone does not provide sufficient problem coverage.

- Since the combination of strategies proves superior for some of the experiments but not for others, we are able to infer a relatively abstract characterization of the classes of problems for which the evolutionary architecture presented here provides substantial performance benefits over its individual components operating in isolation. In particular, problems in which there are strong interactions among goals often provide serious challenges to general purpose reasoners. Specialized agents typically provide efficient resolution of these goal interactions, and model-based adaptation can leverage this strategic advantage.

Thus this research demonstrates that for a substantial class of problems, the combination of approaches allows an agent to adapt to novel functional demands which are beyond the capabilities of simple model-based reasoning and are prohibitively time-consuming to solve by either generative planning or reinforcement learning. These results indicate that this combination *does* provide substantial benefits over any of its elements alone.

References

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

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

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

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

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

[Genesreth & Fikes, 1991] Genesreth, M. R. & Fikes, R. (1991). *Knowledge Interchange Format Version 2 Reference Manual*. Stanford University Logic Group.

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

[Goel et al., 1997a] Goel, A. K., Beisher, E., & Rosen, D. (1997a). Adaptive process planning. Poster Session of the 10th International Symposium on Methodologies for Intelligent Systems.

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

[Goel & Murdock, 1997] Goel, A. K. & Murdock, J. W. (1997). Meta-cases: Explaining case-based reasoning. In I. Smith & B. Faltings (Eds.), *Proceedings of the Third European Workshop on Case-Based Reasoning* Lausanne, Switzerland: Springer.

[Goodwin, 1994] Goodwin, R. (1994). Reasoning about when to start acting. In K. Hammond (Ed.), *Proceedings of the 2nd International Conference on AI Planning Systems* (pp. 86–91). Menlo Park, California: American Association for Artificial Intelligence American Association for Artificial Intelligence.

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

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

[Kautz & Selman, 1996] Kautz, H. & Selman, B. (1996). Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence - AAAI-96* Portland, Oregon.

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

[Lenat & Guha, 1990] Lenat, D. & Guha, R. (1990). *Building Large Knowledge Based Systems: Representation and Inference in the CYC Project*. Addison-Wesley.

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

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

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

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

[Nau et al., 1999] Nau, D. S., Cao, Y., Lotem, A., & Munoz-Avila, H. (1999). SHOP: Simple hierarchical ordered planner. In D. Thomas (Ed.), *Proceedings of the 16th International Joint Conference on Artificial Intelligence - IJCAI-99* (pp. 968–975).: Morgan Kaufmann Publishers.

[Penberthy & Weld, 1992] Penberthy, J. S. & Weld, D. (1992). UCPOP: A sound, complete, partial-order planner for adl. In *Third International Conference on Knowledge Representation and Reasoning - KR-92* Cambridge, MA.

[Rugaber et al., 1995] Rugaber, S., Stirewalt, K., & Wills, L. M. (1995). The interleaving problem in program understanding. In *Proceedings of the Second Working Conference on Reverse Engineering* (pp. 166–175). Toronto, Ontario, Canada: IEEE Computer Society Press.

[Russell & Wefald, 1991] Russell, S. & Wefald, E. (1991). *Do the right thing: Studies in limited rationality*. Artificial Intelligence. Cambridge, Mass: MIT Press.

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

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

[Stroulia & Sorenson, 1998] Stroulia, E. & Sorenson, P. (1998). Functional modeling meets meta-CASE tools for software evolution. In *Proceedings of the International Workshop Software Program Evolution*.

[Sutton, 1988] Sutton, R. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3, 9–44.

[Tate, 1977] Tate, A. (1977). Generating project networks. In R. Reddy (Ed.), *Proceedings of the 5th International Joint Conference on Artificial Intelligence - IJCAI-77* (pp. 888–893).: William Kaufmann.

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

[Weld et al., 1998] Weld, D. S., Anderson, C. R., & Smith, D. E. (1998). Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the 15th National Conference on Artificial Intelligence - AAAI-98* (pp. 897–904).: AAAI Press.