An Investigation into Reactive Planning in Complex Domains

R. James Firby
Department of Computer Science
P.O. Box 2158 Yale Station, New Haven CT 06520

Abstract

A model of purely reactive planning is proposed based on the concept of reactive action packages. A reactive action package, or RAP, can be thought of as an independent entity pursuing some goal in competition with many others at execution time. The RAP processing algorithm addresses the problems of execution monitoring and replanning in uncertain domains with a single, uniform representation and control structure. Use of the RAP model as a basis for adaptive strategic planning is also discussed.[1]

I. Introduction

Automatic planning research has been concerned primarily with the generation of a complete list of actions to carry out a given set of goals. For many domains, particularly those created artificially as in a laboratory or on a factory floor, it makes sense to construct a detailed plan well in advance of execution because the situations expected can be anticipated and controlled. However, it is becoming clear that in more dynamic worlds, where agents exist whose actions cannot be anticipated, the situation at execution time cannot be controlled, and detailed plans cannot be built in advance. As one would expect, the solution to this difficulty is to leave some, most, or even all of the planning to take place during execution when the situation can be determined directly. Systems that build or change their plans in response to the shifting situations at execution time are called reactive planners.

The choice of which detailed actions to put in a plan usually depends on the context in which they will be executed. If that context cannot be computed in advance then the actions cannot be chosen appropriately. For example, planning the arm motions for the loading and unloading portions of a delivery task is both pointless and impossible before the cargo and the loading docks have been examined. More generally, having to choose actions at execution time is unavoidable in any domain where there is uncertainty about what will be encountered after an action is executed. Such uncertainty arises when independent agents or processes can change the world, when actions might not work exactly right, or when there are just too many interacting variables involved in predicting the future.

Reactive planning concerns itself with the difficulties of direct interaction with a changing world and must confront many of the outstanding issues from conventional, strategic planning research. In particular, the problems of execution monitoring and low-level replanning cannot be avoided when constructing a reactive planner. The world state must be monitored continually at execution time if actions are to be chosen based on that state. Furthermore, if the system is to adapt to any situation encountered on the way to a goal, selecting the next step in the plan becomes indistinguishable from changing the plan because of a problem with the last step. Problems will make themselves apparent in the new world state and choosing an appropriate next step will automatically take them into account.

This paper describes an investigation into reactive planning that takes the extreme position of using no prediction of future states at all. Plan selection is done entirely at execution time and is based only on the situation existing then. This approach was chosen, not because an extreme system would be a good planner by itself, but because reactive plan execution must occur at some level in every system; the static action list generated by previous planners lacks the flexibility to confront the dynamic domains of current interest. By studying the problems of reactive plan execution without the complexities of lookahead, this study strives to define a form and content for the representation of more adaptive plans. A traditional strategic planner working with this representation should exhibit a more robust behavior than is possible with static actions lists.

A. Related Work

A great deal of research has been done in the field of planning and good reviews of this work exist in [Joslin and Roach, 1986] and [Chapman, 1985]. In general these investigations have examined the problem of constructing a fixed, static plan for a highly predictable world in which no sensory feedback is required at execution time. The problem of verifying that the execution of such a plan unfolds as expected in a less predictable domain was recognized early and discussed by Sacerdoti [Sacerdoti, 1975] (among others), and recently researchers have been attacking the

[1]The work reported in this paper was supported in part by the Defence Advanced Research Projects Agency under Office of Naval Research contract N00014-85-K-0301

problem with systems that add sensory verification activities to otherwise static plans. Brooks [Brooks, 1982] uses models of domain uncertainty and expected error accumulation to decide where to insert monitoring tasks, while Gini [Gini *et al.*, 1985] uses a model of planner intent to decide when expected situations should be verified. Doyle [Doyle *et al.*, 1986] presents a system for inserting sensory verification tasks into a plan to check that expected world states really hold and Tate [Tate, 1984] has also cast some light on this subject. All of these systems assume that planning involves putting together only effector actions and that sensor actions should be spliced into the plan afterwards as seems appropriate.

Wilkins [Wilkins, 1985] approaches the problems in plan execution by looking at what to do when execution verification shows that something has gone wrong. His work on error recovery concentrates on defining a plan representation that facilitates determination of the parts of a plan which have been compromised by a failure and therefore need to be rethought. Fox and Smith [Fox and Smith, 1984] have discussed the problems of plan failure and replanning in the context of shop floor scheduling.

Miller [Miller, 1985] assumes that basic plan representations must include sensory operations as well as effector actions. In contrast to adding sensory tasks after plan construction, his model builds explicit sensory tasks into the plan right from the beginning. Miller's system uses a scheduling algorithm to integrate all sensor and effector tasks into a single coherent plan before execution. Except for certain limited types of servo correction however, execution time verification and replanning are not dealt with after the initial plan has been constructed. The FORBIN planner discussed in [Firby *et al.*, 1985] builds further on this work.

A markedly different approach to planning has been put forward by Chapman and Agre [Chapman and Agre, 1986] based on the idea of concrete situated activity. Their idea is essentially one of purely reactive planning organized around situation-action like rules. Direct sensory input is used to index structures suggesting possible subsequent actions. Instead of using sensors sparingly to verify constructed plans, sensors must always be active to supply the concrete information on which to base action decisions. Complex activity arises from the continual activation of appropriate actions with no anticipation of the future.

B. The Reactive Action Package

The reactive planner described in this paper is based on the idea of reactive action packages or RAPs. A RAP is essentially an autonomous process that pursues a planning goal until that goal has been achieved. If the system has more than one goal there will be an independent RAP trying to accomplish each one. Each RAP obeys three principles while it is running. First, all decisions of what action to execute next in pursuit of a goal must be based only on the current world state and not on anticipated states. Second, when a RAP finishes successfully, it is guaranteed to have satisfied its goal and to have executed all sensor actions required to confirm that success. Third, should a RAP finish *without* achieving its goal, it will have exhausted every possible avenue of attack; a RAP will fail only if it does not know any way to reach its goal from the current state.

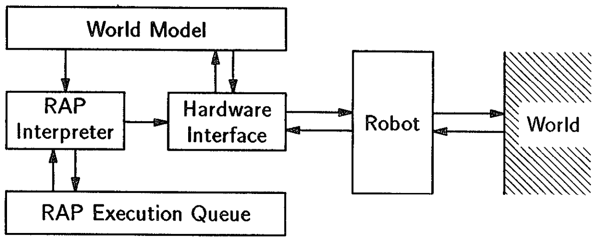

Figure 1: The RAP Execution Environment

To adhere to these principles, a RAP planner must come to grips with the problems of execution monitoring and low-level replanning. Execution monitoring is required to maintain an up-to-date current world model. Every action executed must return some form of feedback about its success or failure to ensure that the world model remains an appropriate basis for planning decisions. Furthermore, some RAPs may need to issue sensor operations in addition to this feedback in order to monitor the progress of their actions in more detail. Some form of low-level replanning must also take place within a RAP to ensure that it explores all approaches to achieving its goal before returning failure. The reactive planner described in this paper consists of a RAP execution environment and processing algorithm that exhibits these characteristics. The planner is used to manage a robot.

II. The Basic RAP Planner

Each RAP should be thought of as an independent entity, pursuing its goal in competition with the other RAPs in the system by consulting the current world state and issuing commands to alter that state. The RAP execution environment shown in figure 1 supports this view of RAP execution. The world model holds the system's understanding of the current world state, the hardware interface controls communication with the real world, and the RAP interpreter and execution queue provide a mechanism for coordinating competition between RAPs. A RAP waits on the execution queue to be selected by the interpreter for its turn to run. When it does run, a RAP consults the world model and issues commands to the hardware interface. The interface passes those commands on to the robot hardware, interprets feedback, such as sensor reports or effector failures, and makes appropriate changes to the world model. Interleaved RAP execution arises when the running RAP stops and returns to the queue to wait for a subgoal to complete and the interpreter chooses another to run in its place (see section III.).

An important aspect of this architecture is the relationship between the world model and the hardware interface. The RAP interpreter must strive to run in real time and therefore all automatic inference within the system must be kept tightly under control. To meet this requirement, the world model remains strictly static: no forward inference is allowed when facts are added or changed and no backward inference is allowed when queries are made. All changes to the model must be handled explicitly by the hardware interface or by the RAPs themselves.

The hardware interface has detailed expectations about the way primitive hardware commands will change the world. It uses this knowledge to interpret the successes and failures returned by actual hardware operations and make appropriate changes to the world model. For example, if a command is issued to grasp a specific object and the hardware returns success, then the interface updates the world model to reflect that the object has been grasped. On the other hand, when the hardware returns some reason for failure, that reason is used to try and straighten out inconsistencies in the world model (*i.e.*, noting that the object is too slippery to grasp, that the gripper is broken, *etc.*). This requires that enough real world feedback come from the hardware to ensure that the interface can maintain a reasonable model of the true world state.

Although it is fair to expect the hardware interface to keep the world model consistent and up-to-date with respect to facts tied closely to direct sensor feedback data, it is unreasonable to assume that it will infer more abstract truths or initiate goals to explain complex failures. Such abstract properties and failures must be derived or explained by the RAPs themselves. For example, a RAP might be responsible for running the dishwasher. Pushing the start button should start the machine and, given appropriate feedback, the hardware interface can update the world model to reflect that the button was pressed. However, before noting in the world model that the dishwasher is running, the "on" light should be checked as confirmation. This sort of high-level knowledge about dishwashers belongs in the dishwasher RAPs and not in the hardware interface charged with monitoring primitive actions. Thus, the dishwasher starting RAP would issue a command to push the start button, a command to check the "on" light and, if both succeeded, would update the world model to reflect that the dishwasher was running.

This division of labor in the RAP execution environment has two desirable characteristics. First, a natural coupling is made between the world model and the real world through hardware level feedback that occurs irrespective of the commands in any particular RAP. Second, any additional complex inference that is required becomes the responsibility of one or more RAPs and thus falls under the same control mechanisms as any other robot activity.

III. Issues in RAP Execution

As a RAP runs and issues commands, it is doing a real-time search through actual world states looking for a path to its goal. In complex domains where general heuristics for

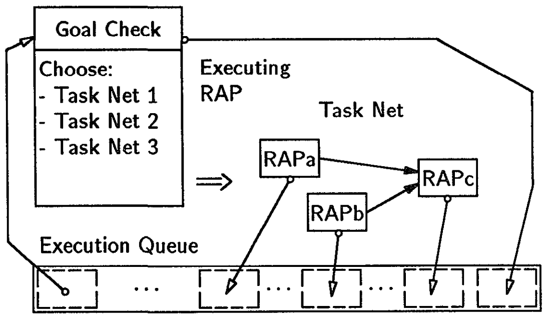

Figure 2: An Illustration of RAP Execution

deciding on the applicability of a given command are not well developed, a simple, blind search from state to state can be very inefficient. To limit the search performed, a RAP holds a predefined set of methods for achieving its goal and only needs to choose between these paths rather than construct new ones. A typical method consists of a partially ordered network of subtasks called a task net and each subtask in the net is either a primitive command or a subgoal that will invoke another RAP. To allow interleaved RAP execution, a RAP runs by consulting the world state, selecting one of its methods and issuing that method's task net all at once. The RAPs and commands in the task net are added to the execution queue and the running RAP is suspended until they have been completed.

This hierarchical style of RAP execution is achieved with the interpreter algorithm illustrated in figure 2. First, a RAP is selected by the interpreter from the RAP execution queue. Selection is based on approaching temporal deadlines and on the ordering constraints placed on RAPs by task nets. If the chosen RAP corresponds to a primitive command it is passed directly on to the hardware. Otherwise the interpreter executes it. As shown in the illustration, each RAP consists of two parts: a goal check and a task net selector. RAP execution always begins with the goal check consulting the world model to see if its goal has already been accomplished. If it has the RAP finishes immediately with success. Otherwise, the RAP tries to choose an appropriate task net. If no net is applicable in the current situation, the RAP must signal failure, but one usually is and the RAP sends it to the execution queue. At this point the RAP has selected a plan for achieving its goal and must wait to see how things turn out. To wait, the RAP is placed back on the execution queue by the interpreter to run again once its task net has finished. When the RAP comes up again, it executes exactly as before. Thus, a RAP keeps choosing task nets until either its goal is achieved, as determined by its goal check, or the world state rules out every task net that it knows about.

This method of specifying and running RAPs allows for a hierarchical and parallel pursuit of RAP goals, but raises the problem of coordination among the different subRAPs in a task net. If an early member of a task net fails, then it is probably pointless to execute those that follow; the method the task net represents simply isn't working out. To deal with this situation, the system keeps track of task net dependencies and removes all the members of a task net from the queue when any one of them fails to achieve its goal.

Another problem with using task nets is that one might fail without changing the world enough to cause the RAP that spawned it to select a different one. In this situation, the RAP will restart, note that its goal has not been satisfied and choose the same task net over again. If nothing intervenes to change the world in some way, such a loop could continue indefinitely. The best solution to this problem is not obvious and is still an area of active investigation. The current system has an execution-time loop detector that flags any RAP that selects the same task net repeatedly without success. Once flagged as a repeat offender, a RAP is given low priority on the execution queue for a while in hope that the world will change. If that doesn't happen, the RAP is eventually made to fail so its parent can try and choose a different task net for its goal. Allowing this tenacious pursuit of goals is a necessary part of dealing with the problem of unintentional interference between competing RAPs.

A. Interactions Between Task Nets

A classic problem with hierarchical planners is that plans for conjunctive goals can clobber each other unintentionally. This problem manifests itself in the RAP planner when an early RAP in a task net sets up a particular state for some later RAP and a third, independent, RAP or process upsets that state. If execution of the task net were to continue after this type of interference, it is likely that the later RAPs would fail and work would have been done for nothing. A standard technique for preventing such wasted work is to place protections on the states established by early RAPs and prevent their change. However, such protections are too restrictive and difficult to manage within the RAP model of execution. One problem is that enforcing a state prevents many useful interactions. For example, one RAP might pick up a glass to move it to the kitchen, and another will want to put it down temporarily to switch the light off before leaving the room. Enforcing something like `(hold glass)` until reaching the kitchen would prevent putting the glass down to free up the hand. A second problem with trying to prevent a state change is that it requires looking into the future and thus violates our goal of building a purely reactive planner. To keep a state unchanged requires asking an action whether it will effect that state before it is executed rather than waiting to see what has changed afterwards. Finally, a state cannot be enforced at all if an external agent or process decides to change it.

The RAP planner deals with interference between task nets without protections. Whenever a task net is chosen, the state that each RAP in the net is designed to establish (if there is one) is attached to the RAP it is being established for. These states form a validity check on the later RAPs which can be evaluated at their execution. After a RAP's goal check, the interpreter checks all states attached to it by earlier RAPs to see if they are still true. If they are, then no interference has occurred and execution of the RAP is still appropriate. If not, an assumption has been violated and the RAP fails causing removal of the task net it belongs to from the execution queue.

B. Uncertainty in the World Model

Another problem that can occur during RAP execution is for the world model to become inconsistent with the state of the real world. This can occur for many reasons including other agents changing the world, simple lack of information about something in the world, or failure to account properly for the evolution of an independent process. Rather than try and deal directly with the uncertainty this causes in the world model, the system just ignores it. When unquestioned faith in the world model results in a primitive command being attempted that cannot possibly succeed, the hardware and hardware interface are supposed to analyze the subsequent failure and correct the world model. For example, if a command is issued to lift a particular rock but it fails because the rock is too heavy, the hardware interface should interpret the failure that way and alter the world model to reflect that the rock is too heavy to lift. Then when the RAP attempting to lift the rock gets around to trying again, it will notice the rock is too heavy and try something else. In this way, the world model is made consistent through corrective feedback from the domain, and replanning required because of previous inconsistency occurs automatically.

C. The RAP Interpreter

In summary, the RAP-based reactive planner described above separates each RAP into three parts: the *Goal Check* and *Task Net Selector* which form the predefined body of the RAP and a *Validity Check* which gets added when the RAP becomes part of a task net. This simple RAP structure is interpreted according to the following algorithm, somewhat reminiscent of the NASL interpreter described in [McDermott, 1976]:

1. Choose a RAP or command to execute from those waiting on the execution queue. If a command is chosen simply pass it on to the hardware and choose again until a RAP comes up.
2. Run the RAP's *Goal Check* to see if its goal has been achieved. If it has then this RAP is finished and should return success.
3. Run the RAP's *Validity Check* if it has one. If the test fails then a task net assumption has been violated, this RAP is no longer appropriate and it should finish returning failure.
4. Run the RAP's *Task Net Selector* to choose a task net to achieve its goal starting from the current world model. If no appropriate task net is known then the

goal of this RAP cannot be achieved and the RAP should finish returning failure.

5. Place the subtasks from the selected net on the execution queue so that they will be run in accordance with the orderings placed on them by the net.
6. Put the RAP back on the execution queue to be run after its task net has finished executing.
7. Go to (1) to choose another RAP.

IV. Summary and Conclusions

The idea of purely reactive planning as typified by the RAP model described in this paper has one obvious shortcoming: it cannot deal effectively with problems that require thinking ahead. Making plan choice decisions based only on the current world state precludes identification and prevention of impending detrimental situations. This can cause pursuit of the planner's goals to be inefficient, unsuccessful, or even dangerous. Some inefficiencies occur because interactions between competing task nets are only repaired and not prevented during RAP execution. Also, RAPs cannot take expected states like rain into account, so choices like leaving the umbrella behind because it's sunny can be short sighted and cause unnecessary trips back once the rain starts. Poor management of scarce resources is a similar inefficiency that can prevent otherwise successful plans from working. Finally, not being able to look ahead can cause disasters which should be preventable, like carrying an oil lantern downstairs to look for a gas leak.

Strategic planning, or looking ahead into the future, is required to detect inefficiencies and unhappy situations before they occur. Given the RAP model of reactive planning, a strategic planner's job would be to put constraints on RAP behavior before execution to either prevent or encourage specific situations. Such constraints might take the form of ordering RAPs on the execution queue or forcing certain RAPs to make particular task net choices. For example, left on its own a RAP might elect to not pick up an umbrella because it is sunny, but the strategic planner, knowing that it will rain, could force the RAP to choose a task net that included taking the umbrella.

In summary, the purely reactive RAP planner discussed in this paper has several important features. It is extremely adaptive and hence tolerant of uncertain knowledge introduced by the actions of other agents or by the inherent complexity of the domain. In addition, by studying the domain feedback required to support RAP processing, the role of execution monitoring has been clarified and integrated as a natural part of the planning and execution environment. Similarly, issues of plan failure and replanning are subsumed by a single, uniform RAP processing algorithm. Finally, it is suggested that strategic planners would achieve more flexibility in many domains by assuming RAP-based plan execution and generating RAPs constrained only as required to prevent serious inefficiencies and dangerous situations.

References

[Brooks, 1982] Rodney Brooks. *Symbolic Error Analysis and Robot Planning.* Technical Report AI Memo 685, MIT, September 1982.

[Chapman, 1985] David Chapman. *Planning for Conjunctive Goals.* Technical Report TR 802, MIT Artificial Intelligence Laboratory, 1985.

[Chapman and Agre, 1986] David Chapman and Philip E. Agre. Abstract reasoning as emergent from concrete activity. In *Workshop on Planning and Reasoning about Action*, Portland, Oregon, June 1986.

[Doyle *et al.*, 1986] Richard Doyle, David Atkinson, and Rajkumar Doshi. Generating perception requests and expectations to verify the execution of plans. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, AAAI, Philadelphia, PA, August 1986.

[Firby *et al.*, 1985] R. James Firby, Thomas Dean, and David Miller. Efficient robot planning with deadlines and travel time. In *Proceedings of the IASTED International Symposium, Advances in Robotics*, IASTED, Santa Barbara, CA, May 1985.

[Fox and Smith, 1984] Mark S. Fox and Stephen Smith. The role of intelligent reactive processing in production management. In *13th Meeting and Technical Conference*, CAM-I, November 1984.

[Gini *et al.*, 1985] Maria Gini, Rajkumar S. Doshi, Sharon Garber, Marc Gluch, Richard Smith, and Imran Zualkenian. *Symbolic Reasoning as a Basis for Automatic Error Recovery in Robots.* Technical Report TR 85-24, University of Minnesota, July 1985.

[Joslin and Roach, 1986] David E. Joslin and John W. Roach. *An Analysis of Conjunctive-Goal Planning.* Technical Report TR 86-34, Virginia Tech, 1986.

[McDermott, 1976] Drew V. McDermott. *Flexibility and Efficiency in a Computer Program for Designing Circuits.* PhD thesis, Dept. of Electrical Engineering, MIT, September 1976.

[Miller, 1985]
David P. Miller. *Planning by Search Through Simulations.* Technical Report YALEU/CSD/RR 423, Yale University Department of Computer Science, 1985.

[Sacerdoti, 1975] Earl D. Sacerdoti. *A Structure for Plans and Behavior.* Technical Report SRI Project 3805, Stanford Research Institute, 1975.

[Tate, 1984] Austin Tate. *Planning and Condition Monitoring in a FMS.* Technical Report AIAI TR 2, University of Edinburgh, Artificial Intelligence Applications Institute, July 1984.

[Wilkins, 1985] David E. Wilkins. Recovering from execution errors in SIPE. *Computational Intelligence*, 1(1), February 1985.