Modularity Issues in Reactive Planning

R. James Firby *
Artificial Intelligence Laboratory
Computer Science Department
University of Chicago
1100 East 58th Street
Chicago, IL 60637
firby@cs.uchicago.edu

Abstract

The RAP reactive plan execution system is specifically designed to accept plans and goals at a high level of abstraction and expand them into detailed actions at run time. One of the key features of the RAP system is the use of hierarchical expansion methods that allow different paths of execution for the same goals in different situations. Another central reason for the hierarchy is to create modular expansion methods that can be used in the execution of many different tasks. However, experience using the RAP system to control the University of Chicago robot Chip in the 1995 IJCAI robot competition has shown that there are difficult trade-offs between modularity and correctness in a predefined plan hierarchy.

This paper describes the RAP hierarchies used to control the robot while cleaning up a small office space and discusses some of the issues raised in writing these RAPs to be useful for other tasks as well. In particular, realistic reactive plans must support concurrency that crosses simple modular decomposition boundaries and efficient strategies for carrying out tasks that depend on active sensing require advanced knowledge of sensing requirements.

Introduction

An autonomous, intelligent agent requires a variety of capabilities. It must be able to sense the world and control its actions in real-time. It must also be able to execute complex, multi-step plans, confirm that plan steps have had their intended effects, and deal with problems and opportunities as they arise. An agent's ability to carry out complex plans in dynamic environments hinges on knowing large numbers of different methods for carrying out a task in different situations and for detecting problems and correcting them.

Modularity and reusability are especially important aspects of any plan library the agent uses to encode these methods. An agent must be able to pursue a variety of different goals at different times and each goal will require different plans. Even a single plan will consist of multiple steps that each corresponds to a new subgoal for the agent. Many of these subgoals will correspond to conceptually identical tasks like picking an object up or moving to a specific location. If it is possible to reuse the same reactive plans for these subgoals, even when they are used as steps in different higher-level plans, the agent's behavior will be much easier to program, much more reliable, and the plans themselves will be easier to adapt and learn.

The Animate Agent Project at the University of Chicago is building an agent control system that integrates reactive plan execution, behavioral control, and active vision within a single software framework. The architecture splits the agent's representation of actions into two levels: one consisting of continuous, real-time active vision and control processes that can be combined in different ways at run-time to create different behavior, and another consisting of hierarchical methods of discrete steps for achieving goals and coping with contingencies in the world. Modularity is emphasized in both levels so that sensing and action processes can be combined in multiple ways and task methods can share common subtasks.

This paper explores some of the difficulties encountered when creating a reactive plan library for the office cleanup task at the 1995 IJCAI robot contest. Our approach was to try and create as general a plan library as possible so that the major components could be reused for other tasks. We found that most of the actions required formed very natural and modular task hierarchies. However, it was also the case that some tasks required tying active sensing tasks very tightly to specific features in the world and maintaining those ties across the natural task boundaries. There seemed to be a trade-off between modularity and effectiveness. This paper discusses those trade-offs and argues that, to be both modular and correct, task methods must be able to spawn new top-level goals.

*This work was supported in part by ONR contract N00014-93-1-0332 and ARPA contract N00014-93-1-1394.

The Animate Agent Architecture

The organization of the Animate Agent Architecture is in two levels. The low level of the architecture is a collection of soft real-time routines called *skills* that can be rearranged into different control loops at different times. At a higher level, the RAP plan execution system manipulates the set of routines running at any given time to create a sequence of control states to accomplish a specific task. Skills are meant to capture those aspects of sensing and action tasks that are best described in terms of continuous feedback loops while RAPs capture those aspects of task execution that are best described as symbolic plans.

Each skill is defined by a separate program and can be enabled or disabled at will. When a skill is enabled it is sent a set of parameters and begins to run. All running skills can access current sensor values and values generated by other skills on global *channels* (Gat 1991). Skills can also set actuator and channel values or constrain the values actuators and channels take on.

Skills are typically enabled in groups that form coherent sets of processes that work together to carry out specific actions in the world. Skills are programmed to generate asynchronous *signals* when they detect that a task is complete or some problem has occurred. These signals are caught by the RAP system and tell it when to move on to the next step in a plan.

The RAP system is essentially a reactive plan interpreter. It takes a set of task goals as input (either as a plan from a planner, or in the case of our experiments as top-level goals) and refines each goal hierarchically until primitive actions are reached. The hierarchical plans that define these expansions are the RAPs that reside in the RAP library.

The RAP system interacts with the skill system using primitive steps that enable and disable skills, and by waiting for signals generated by enabled skills. RAPs are usually structured to enable a meaningful set of skills concurrently and then wait for that set of skills to signal successful or unsuccessful completion. More detail on the RAP system can be found in (Firby 1989; 1994).

The Office Cleanup Task

One of the tasks at the 1995 IJCAI Robot Competition was to pick up all of the trash on an office floor and sort it into trash and recycling bins. The contest defined soda cans to be recyclable and paper cups to be trash. The goal was to correctly sort as many pieces of trash as possible in a given amount of time. In building skills and RAPs for this task, our goal was, and continues to be, to try to be as modular as possible so they can be used for other tasks as well. We have attempted to make as little of our task representation as possible trash specific.

Cleanup Skills

One approach we have been using to make skills more generic is to designate objects in the world indexical-functionally as the output of tracking skills (Agre & Chapman 1987). For example, we do not have a skill specific to picking up a piece of trash. Instead, there is a skill to pick up a small object that is being designated by another skill writing to the `target-location` channel. We can then pick up a variety of object types simply by choosing different visual skills to designate their `target-location`.

The following visual skills are used in the cleanup task and are described in more detail in (Firby *et al.* 1995):

- FIND-SMALL-OBJECTS locates and classifies all of the small object on the floor in the current visual scene.
- FIND-HAUSDORFF-MATCH locates an area in the image that matches a defined edge model using the hausdorff distance to rank possible matches.
- TRACK-SMALL-OBJECT runs continuously, tracking the location of a designated small object in the visual scene.
- TRACK-HAUSDORFF runs continuously, applying the Hausdorff matcher repeatedly to track the location of a known object in the scene.

In addition to finding and tracking objects, the cleanup task required skills for moving the robot:

- MOVE-TO-TARGET moves the robot to the (x, y) coordinate defined by the `target-location` channel while avoiding obstacles.
- ORIENT-TO-TARGET orients the robot with respect to (x, y) coordinate defined by the `target-location` channel so that the arm can reach the location.
- Move directly forward or backward a given amount.

The cleanup task also uses several skills for moving the robot's arm and gripper. These skills are very simple because CHIP has a simple four degree of freedom arm that extends outward in front of its body.

Cleanup RAPs

The interface between the RAP system and the skill system is in the form of RAP methods that enable and disable skills. For the cleanup task, this includes primitive RAPs for:

- Looking for objects.
- Tracking objects.

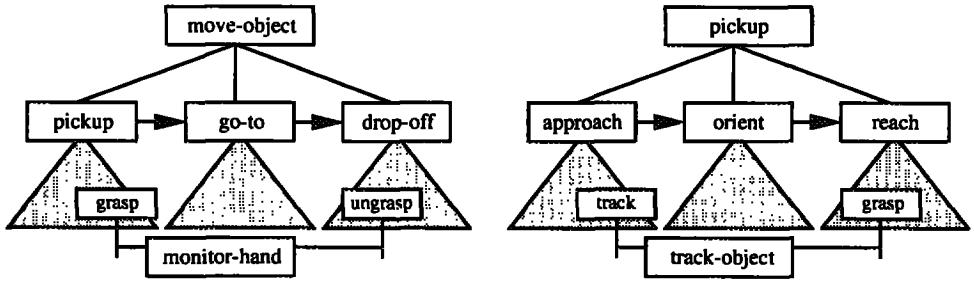

Figure 1: Examples of Overlapping Actions

- Moving the pan/tilt head.
- Moving to a target location.
- Orienting to a target location.
- Moving the arm and wrist and gripper.
- Moving forward and backward.

Running these RAPs in parallel allows the construction of a variety of different concurrent skill sets. For example, approaching a piece of trash requires starting up a tracking skill in parallel with a skill to move toward the tracked object. Similarly, orienting to an object so it can be picked up requires tracking the object in parallel with the skill for orienting to the tracked object.

A hierarchy of more abstract RAPs is built on top of the primitives. The hierarchy is defined by the subgoals that we believe will be generally useful for other tasks in the future. Only at the very top of the hierarchy does information specific to trash cleanup come into play.

The RAP hierarchy specifies general methods for:

- Finding an object.
- Moving to an object.
- Orienting to an object.
- Picking an object up.
- Dropping an object off.

For example, a simplified method from the RAP for approaching the current object of interest is shown below:

```
(define-rap (go-to-object ?radius)
 (succeed (near-object ?radius))
 ...
 (method
  (context (current-object ?type ?x ?y))
  (task-net
   (sequence
    (t1 (pan-to (angle-to ?x ?y)))
    (parallel
     (t2 (track-object ?type ?x ?y)(until-end t4))
     (t3 (pan-to-target) (until-end t4))
     (t4 (go-to-target)))))))
```

This RAP first points the camera in the direction of the object to be approached. Then, three sub-tasks are generated to run concurrently: TRACK-OBJECT selects and starts up a tracking skill to generate a `target-location` to the object being looked at, PAN-TO-TARGET starts up a skill to keep the pan/tilt head pointed in the direction of the target location, and GO-TO-TARGET moves the robot towards the target location while avoiding obstacles. The RAP for orienting to an object is very similar. It is particularly important that the plans for approaching and orienting to an object track the object as the robot is moving. Continually tracking a target to close the visual feedback loop is an essential error reduction step.

Finally, at the top level of the RAP hierarchy, RAPs are defined for more cleanup specific behaviors like moving a piece of trash from the floor to an appropriate trash can and systematically searching the room for all pieces of trash.

Thus, the RAPs in the cleanup domain form a natural hierarchy that supports the reuse of subtask plans and hides many of the details involved in executing those subtasks. Picking an object up, or moving an object from one place to another, are relatively abstract RAPs in the hierarchy but are excellent operators for use in other RAPs or plans. Unfortunately, the modularity of the hierarchy also makes it difficult to describe some subtask compositions.

Modularizing Overlapping Actions

A major problem in creating the cleanup RAP hierarchy is the need, on occasion, to instantiate tasks that stretch across natural subtask boundaries. Consider two simple actions from the trash cleanup domain: moving an object from the floor to a trash can, and approaching an object and picking it up. The MOVE-OBJECT task very naturally breaks down into three subtasks: pickup the object, go to the trash can, drop the object in. Similarly, the PICKUP-OBJECT task

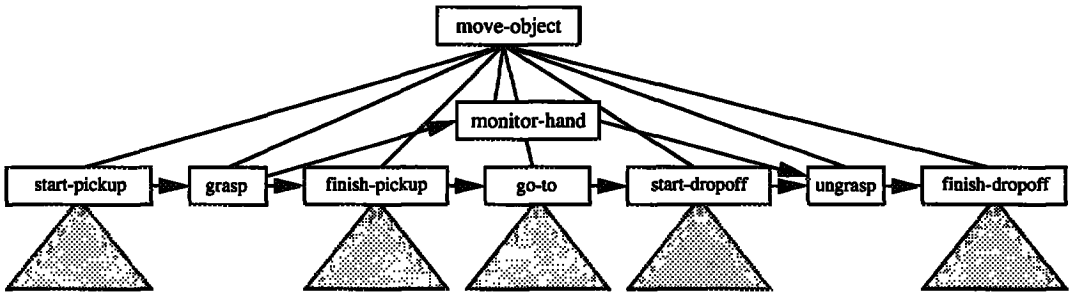

Figure 2: Incorporating Overlapping Tasks

breaks down into three subtasks: approach the object, orient to bring the gripper to the object, and reach out to grasp the object.

In both of these examples, the natural breakdown of the tasks into subtasks is important because those subtasks can be, and are, used to satisfy a variety of other goals as well. However, in both examples, another subtask is required that spans some of the natural breakpoints. As shown in Figure 1 MOVE-OBJECT requires a task to monitor the fact that the gripper is holding the trash and to report a problem if it is dropped. Similarly, PICKUP-OBJECT requires continually tracking the object during approach, orienting, and grasping.

The problem is simple: How can we write RAP methods for the reusable steps in MOVE-OBJECT and PICKUP-OBJECT so they are also compatible with the overlapping subtasks needed to complete the methods PICKUP-OBJECT and MOVE-OBJECT themselves?

Explicit Parallelism

The most direct way to incorporate overlapping tasks into methods is to explicitly encode the required parallelism into the task decomposition. For example, one might encode MOVE-OBJECT to do PICKUP, then GO-TO in parallel with MONITOR-HAND, and finally DROP-OFF. Specifying a concurrent task to monitor a state in this way is similar to the *policy* construct in the RPL language (McDermott 1992; 1991).

Unfortunately, to get the correct overlap for the MONITOR-HAND task, its end-points must be pushed down the task hierarchy, as shown in Figure 2, below the level of abstraction where pickup, go-to, and drop-off are modular units. The resulting reorganization of subtask structure under MOVE-OBJECT (and the similar one required for PICKUP-OBJECT) destroys the modularity inherent in the natural subtask decomposition of the trash cleanup domain.

Parallelism is the representational goal, but directly encoding it into higher-level methods is incompatible with maintaining useful modular subtasks.

Naive Decomposition

Another solution for dealing with subtasks that extend across natural decomposition boundaries is to cut them into non-overlapping pieces. For example, we can cut the MONITOR-HAND subtask in the decomposition for MOVE-OBJECT into three pieces as shown in Figure 3. The pickup subtask uses a MONITOR-HAND subtask in parallel with lifting the arm up, the go-to subtask turns into a carry subtask that includes a MONITOR-HAND in parallel with the go-to, and the dropping off subtask includes MONITOR-HAND until the

```
(define-rap (carry)
  ...
  (parallel
    (t1 (monitor-hand) (until-end t2))
    (t2 (go-to))))

(define-rap (pickup)
  ...
  (sequence
    (t1 (position-arm down))
    (t2 (grasp))
    (parallel
      (t3 (monitor-hand) (until-end t4))
      (t4 (position-arm up)))))
```

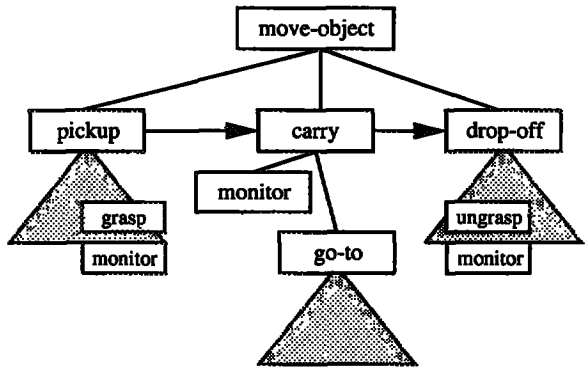

Figure 3: Dividing an Overlapping Task into Pieces

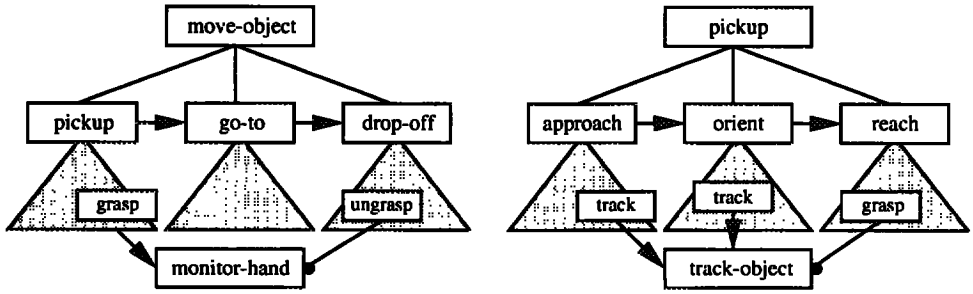

Figure 4: Spawning Monitoring Tasks

object is ungrasped. Using this decomposition, pickup, carry, and drop-off become separate tasks that can be composed in sequence *without* worrying about a separate MONITOR-HAND subtask. The problem of overlapping tasks has gone away.

This solution to the problem often works quite well (and was used on our robot at the actual competition) but it won't work in general. In particular, applying this solution to the PICKUP-OBJECT task requires cutting the visual tracking task into three pieces: one to be part of the approach task, one to be part of the orient task, and one to be part of the grasp task. Splitting up a visual tracking task in this way can cause tracking problems at task boundaries. When everything in the world is stationary, the final state of one tracking task can be fed into the next to maintain continuity (*i.e.*, perhaps the image centroid of the object being tracked as last seen by the orient task might be passed to the tracking component of the grasp task). However, when the object to be grasped is moving or the world is changing quickly, maintaining a positive visual lock on the desired object requires continuous tracking. Stopping and starting the tracker at task boundaries creates a lack of continuity that can cause target loss.

The Real Problem

It is important to understand at this point that the problem of encoding overlapping tasks while maintaining modularity is not a planning problem. The issue is not how, or whether, to generate a task for monitoring the hand or for tracking an object, but rather how to properly describe the scope of such a task. In particular, we are interested in encoding tasks that are conceptually concurrent at a high level of abstraction, but which must be synchronized with subtasks hidden much further down in the hierarchy. Deciding what to monitor and detecting and correcting interactions between independent RAP tasks are independent issues.

Neither of the two simple solutions to the problem of encoding overlapping steps described above — the division of overlapping steps into pieces that match the other subtasks, and the use of explicit parallelism to directly represent the overlapping required — is completely adequate. The only way to correctly represent the required task combinations is to encode the dynamic creation of independent goals.

Spawning Tasks

To correctly solve the problem of including overlapping tasks in modular plans, the method description semantics must be extended to allow the creation of independent subtasks. With this ability, the MOVE-OBJECT method can be encoded as shown in Figure 4. The subtasks for picking up the object, going to the destination, and dropping the object off are all treated as independent. Down within the expansion of PICKUP-OBJECT, after the object is grasped, a MONITOR-HAND is always spawned to watch for the object to be dropped. This task lives on after the PICKUP-OBJECT step completes, running concurrently with going to the destination (or any other task that might be attempted while holding the object) and the first part of dropping the object off. Then, down within DROP-OFF-OBJECT, when the object is released, the MONITOR-HAND task is terminated.

Issues in Spawning Independent Tasks Allowing methods to spawn and terminate independent tasks provides a way to scope the lifetime of concurrent tasks over modular subtask boundaries. However, there are additional details required to make things work correctly. First, a spawned task needs to be named so that it can be referred to by later, independent methods. For example, the subtask under MOVE-OBJECT for dropping an object off must be able to terminate the MONITOR-HAND task started by the completely independent pickup task. Similarly, the subtasks under PICKUP-OBJECT are supposed to be generic tasks for approaching and object, orienting to it, and reaching out to grasp it. All three require a tracking task to be running and hence need to start one if none already

exists (if the orient task is used as the first step in a method for example). To prevent the creation of multiple tracking tasks, each of these subtasks must be able to check to see whether an appropriate tracking task is already running. A named global tracking task makes this check possible.

Second, there must be a way to tie the spawned task to the overall goal it is part of so that goal can be told about problems with the spawned task. For example, suppose the gripper drops a piece of trash while the robot is going to the trash can. The spawned MONITOR-HAND task can be written to notice this problem and stop the robot to pick the trash up again. However, what should happen if the trash has rolled under a table and cannot be located? The MONITOR-HAND task must be able to signal this problem to the MOVE-OBJECT task so it can deal with the failure.

Introducing constructs into the RAP language for spawning and naming tasks, terminating tasks, and intercepting signals gives the method fragments shown in Figure 5 for the MOVE-OBJECT task. The method for MOVE-OBJECT uses three steps and during those steps will fail if a `(lost-object)` signal is generated. Within the PICKUP-OBJECT method, a MONITOR-HAND task named `?task` is created and a notation is made in RAP memory. Within the DROP-OFF-OBJECT RAP a method is included that checks whether there is a monitoring task, grabs its name from memory, and then terminates it just before the object is ungrasped. Notice that the PICKUP-OBJECT and DROP-OFF-OBJECT remain self-contained in achieving their primary goals. PICKUP-OBJECT does not care how the MONITOR-HAND task is used or what it does, and the DROP-OFF-OBJECT task drops the object whether the hand is being monitored or not. Notice also that there is no need for a separate "carry" goal, the go to destination task can be used directly because the MONITOR-HAND task is spawned by the pickup method. The hand monitor task is created and terminated exactly when it should be: when an object is grasped and ungrasped. These RAPs together implement a policy without collapsing the task hierarchy.

Orphaned Tasks A final complication in allowing spawned tasks is what to do should the MOVE-OBJECT method fail for some reason such as the destination being unreachable. In this case, the subtasks explicitly used in the method will go away, (*i.e.*, PICKUP-OBJECT, GO-TO, and DROP-OFF-OBJECT) but the independent spawned MONITOR-HAND task will be left because it is essentially unknown to the MOVE-OBJECT method (except that MOVE-OBJECT is encoded to expect a signal from it). This anonymity is what leaves the other subtasks modular and reusable and it cannot be abandoned. Furthermore, it makes a certain amount of sense for the MONITOR-HAND task to persist as long as the hand is holding the object, even if it is no longer holding it for a purpose.

```
(define-rap (move-object ?container)
  ...
  (on-event (lost-object) :fail
    (sequence
      (t1 (pickup-object))
      (t2 (go-to ?container))
      (t3 (drop-off-object)))))

(define-rap (pickup-object)
  ...
  (sequence
    (t1 (start-pickup))
    (t2 (grasp))
    (t3 (spawn (monitor-hand) ?task)
        (mem-add (monitoring-hand ?task)))
    (t4 (finish-pickup))))

(define-rap (drop-off-object)
  ...
  (method
    (context (monitor-hand ?task))
    (task-net
      (sequence
        (t1 (start-drop-off))
        (t2 (ungrasp))
        (t3 (terminate ?task))
        (t4 (finish-drop-off)))))
  (method
    (context (not (monitor-hand ?task)))
    (task-net
      (sequence
        (t1 (start-drop-off))
        (t2 (ungrasp))
        (t3 (finish-drop-off))))))
```

Figure 5: RAP Methods for the Move-Object Task

So, the easiest solution is for the MONITOR-HAND task to live on until the object is actually put down, at which point the DROP-OFF-OBJECT task (or any other put down task) will explicitly terminate it.

The TRACK-OBJECT subtask of PICKUP-OBJECT has a somewhat different semantics. In this case, the subtask is supposed to maintain a continuous designation of the object being tracked for the subtasks to use. Unlike the MONITOR-HAND subtask, there is little point in the tracking task continuing to run after the tasks needing its information have gone away. Worse, when a new task requiring the vision system begins to run, the orphaned tracking task may interfere. Within the RAP system it is possible to get around this problem by careful use of tracking task names (so that an old tracker will not be assumed to work for a new object), specification of a software "tracking resource" that new

tracking tasks take from old ones causing the old ones to terminate, and encoding tracking tasks to terminate when they lose track of their target. However, the right solution is to include more generic goal knowledge to terminate orphan tracking tasks when the higher-level tasks that caused their creation go away.

Spawning Tasks and RAP Semantics

The discussion above argues that the only way to create modular task descriptions that *also* include overlapping concurrent tasks to designate and monitor features in the world is to include the ability to spawn independent, named tasks in the task description language. Without this ability, there is no way to create the appropriate monitoring or tracking tasks at exactly the points where the states involved become true deep within generic task decompositions. Task expansion methods must be able to create, terminate, and refer to other tasks without regard to where those tasks might have come from.

Giving the RAP system the ability to refer to and alter the overall task state in this way is a significant change in the expressive power of the RAP language. As a strictly hierarchical language, RAP methods can create task expansions of arbitrary complexity, but the scope of the subtasks in a method is well defined and they have no effect on other methods a RAP task can manipulate only the subtasks it spawns. However, giving methods the ability to alter the top-level goals of the system (by spawning and terminating top-level tasks) means that RAP methods can control the way that RAP expansion proceeds.

Allowing RAP methods to spawn new, independent tasks is required to make subtask descriptions more modular so they are easier to combine. However, care must be taken because allowing new tasks to be spawned in this way can make the semantics of RAP representations considerably more obscure.

Opportunistic Task Creation

Another problem encountered while developing RAPs and skills for the IJCAI office cleanup task is that of knowing when to do passive visual sensing to support goals that are not yet active. In particular, while the robot looks for trash, it would be nice, and more efficient, if it was also looking for trash cans with whatever spare processing resources are available. Similarly, when looking for trash cans it would be nice to notice trash.

This problem can be solved by simply writing the SEARCH-FOR-TRASH task to run a parallel task to WATCH-FOR-TRASH-CANS and vice versa. However, this solution overlaps specific tasks in a very non-modular way. It would be much more useful for the WATCH-FOR-TRASH-CANS task to be generated automatically when trash cans will be relevant in the future.

For example, consider the schematic RAP fragment for throwing away a piece of trash shown below:

```
(define-rap (toss-out-trash)
  ...
  (sequence
    (t1 (search-for-trash))
    (t2 (pickup-object))
    (t3 (search-for-trash-can))
    (t4 (go-to-object))
    (t5 (drop-off-held-object))))
```

This method shows a relatively modular breakdown of the task of tossing out a single piece of trash. However, the simple, sequential breakdown of the task does not take into account the potential efficiencies of noting the locations of trash cans seen while looking for trash. When step `t1` is running, `t3` simply waits its turn and no visual processing effort is put into looking for trash cans.

In general, anticipating opportunities is a planning problem that requires looking into the future, noting expected sensing needs, and then creating a task plan that includes early, opportunistic sensing steps. However, it is possible to extend the RAP system to exploit such opportunities in certain limited situations.

One would like to add a rule to the system of the form: When a RAP method includes the step SEARCH-FOR-TRASH-CAN, add a step WATCH-FOR-TRASH-CAN in parallel with the whole method. Assuming WATCH-FOR-TRASH-CAN is basically a passive visual task that examines images "in the background," applying this rule when the TOSS-OUT-TRASH method is instantiated will add a task to the expansion that won't interfere with other steps in the method and might save some time later by noticing a trash can that would otherwise have been missed. The PRS system supports such rules as meta-KAs (Georgeff, Lansky, & Schoppers 1986).

Similar rules can be used to address some of the problems with overlapping tasks discussed in the previous section. For example, one might want a rule that says: Whenever a GRASP action is taken *as part of* the expansion of MOVE-OBJECT, a MONITOR-HAND task should be created. This could be combined with a rule that says: Whenever an UNGRASP action is taken while a MONITOR-HAND task is running, terminate the MONITOR-HAND. The result will be the same behavior as methods containing steps that explicitly spawn and terminate the MONITOR-HAND task.

More generally, rules of this sort are a way to specify modifications to the RAP system's current set of goals outside of the usual constraints of hierarchical expan-

sion. They are a simple way of modifying the tasks in a method based on anticipated interactions with other tasks the system knows about. They can also be seen as a way of specifying *patches* to RAP methods without rewriting the methods themselves. The concept of programming reactive methods and then adding patches as difficult situations are encountered is discussed by Simmons in (Simmons 1994).

A mechanism for defining RAP system rules to be invoked when specific task expansion patterns occur has not yet been completely defined. However, the need to recognize and exploit opportunities like the passive sensing example above along with the power in defining modifications and patches outside of the normal expansion hierarchy has motivated its development.

Conclusions

This paper discusses two problems encountered while constructing RAP task descriptions for an office cleanup task. One problem is the requirement that some tasks monitoring states in the world extend continuously across otherwise modular subtask boundaries. The other problem is anticipating task interactions and opportunities and altering the RAP system's goal structure in response.

Two solutions to these problems were discussed and both acknowledge that there must be a mechanism in the RAP language for creating new top-level tasks within expansion methods. It must be possible to spawn tasks at the point in an expansion when they become relevant relevant and then let them run continually across succeeding independent steps.

The first solution is to augment the existing RAP language to support the creation and termination of independent, named tasks. Coupling these tasks with the ability to send and intercept signals allows a method to be built from steps that start and stop auxiliary tasks and can catch messages from those tasks. Using these constructs it is simple to encode both the MOVE-OBJECT task to continuously watch for the object to be dropped, and the PICKUP-OBJECT task to continuously track the object while it lines up and grasps. With this approach, subtasks must explicitly know when and what auxiliary tasks to spawn and terminate without direct knowledge of what sort of expansion they are taking part in.

The second solution is to create a new language for rules that watch for patterns of task instantiation and world state to occur and then create or terminate additional tasks. This approach acknowledges that all tasks are executed in the context of other tasks, and that one of the main problems in creating modular subtasks is sorting out task interactions. The way to achieve a goal in the world is to begin executing a set of cooperating tasks that can be changed via task rules as execution proceeds. In this situation, odd circumstances, like what to do if an object is dropped or a tracked object is lost, can be seen as patches to the current active goal structure.

The first solution is currently being implemented in the RAP system. The second is being investigated in the more general context of incorporating a planner with flexible goals.

These results suggest that any reactive plan execution system that desires to hide complexity from a planner using a hierarchy of modular subtasks *must* be able to manipulate its own internal tasks and task states. A task plan must be able to create new tasks that are unconstrained by the current task refinement hierarchy, and to refer to and act on any task within the system, not just those created as subtasks. Without these abilities, task plans cannot be made modular at appropriate levels of abstraction and there is no way to shield higher level systems from the details of interacting with the world.

References

Agre, P. E., and Chapman, D. 1987. Pengi: An implementation of a theory of activity. In *Sixth National Conference on Artificial Intelligence*, 268–272. Seattle, WA: AAAI.

Firby, R. J.; Kahn, R. E.; Prokopowicz, P. N.; and Swain, M. J. 1995. An architecture for vision and action. In *Fourteenth International Joint Conference on Artificial Intelligence*, 72–79.

Firby, R. J. 1989. Adaptive execution in complex dynamic worlds. Technical Report YALEU/CSD/RR #672, Computer Science Department, Yale University.

Firby, R. J. 1994. Task networks for controlling continuous processes. In *Second International Conference on AI Planning Systems*, 49–54.

Gat, E. 1991. *Reliable Goal-Directed Reactive Control of Autonomous Mobile Robots*. Ph.D. Dissertation, Computer Science and Applications, Virginia Polytechnic Institute.

Georgeff, M. P.; Lansky, A. L.; and Schoppers, M. J. 1986. Reasoning and planning in dynamic domains: An experiment with a mobile robot. Tech Note 380, AI Center, SRI International.

McDermott, D. 1991. A reactive plan language. Technical Report YALEU/CSD/RR #864, Yale University Department of Computer Science.

McDermott, D. 1992. Transformational planning of reactive behavior. Technical Report YALEU/CSD/RR #941, Yale University Department of Computer Science.

Simmons, R. 1994. Becoming increasingly reliable. In *Second International Conference on AI Planning Systems*, 152–157.