In *Proceedings of the 1998 National Conference on Artificial Intelligence.* Madison, Wisconsin, 1998.

Managing Multiple Tasks in Complex, Dynamic Environments

Michael Freed
NASA Ames Research Center
mfreed@mail.arc.nasa.gov

Abstract

Sketchy planners are designed to achieve goals in realistically complex, time-pressured, and uncertain task environments. However, the ability to manage multiple, potentially interacting tasks in such environments requires extensions to the functionality these systems typically provide. This paper identifies a number of factors affecting how interacting tasks should be prioritized, interrupted, and resumed, and then describes a sketchy planner called APEX that takes account of these factors when managing multiple tasks.

Introduction

To perform effectively in many environments, an agent must be able to manage multiple tasks in a complex, time-pressured, and partially uncertain world. For example, the APEX agent architecture described below has been used to simulate human air traffic controllers in a simulated aerospace environment (Freed and Remington, 1997). Air traffic control consists almost entirely of routine activity; complexity arises largely from the need to manage multiple tasks. For example, the task of guiding a plane to landing at a destination airport typically involves issuing a series of standard turn and descent authorizations to each plane. Since such routines must be carried out over minutes or tens of minutes, the task of handling any individual plane must be periodically interrupted to handle new arrivals or resume a previously interrupted plane-handling task.

Plan execution systems (e.g. Georgoff and Lansky, 1988; Firby, 1989; Cohen et al., 1989; Gat, 1992; Simmons, 1994; Hayes-Roth, 1995; Pell, et al., 1997), also called *sketchy planners*, have been designed specifically to cope with the time-pressure and uncertainty inherent in these kinds of environments. This paper discusses a sketchy planner called APEX which incorporates and builds on multitask management capabilities found in previous systems.

Multitask Resource Conflicts

The problem of coordinating the execution of multiple tasks differs from that of executing a single task because tasks can interact. For example, two task interact benignly when one reduces the execution time, likelihood of failure, or risk of some undesirable side effect from the other. Perhaps the most common interaction between routine tasks results from competition for resources.

An agent's cognitive, perceptual, and motor resources are typically limited in the sense that each can normally be used for only one task at a time. For example, a task that requires the *gaze* resource to examine a visual location cannot be carried out at the same time as a task that requires gaze to examine a different location. When separate tasks make incompatible demands for a resource, a *resource conflict* between them exists. To manage multiple tasks effectively, an agent must be able to detect and resolve such conflicts.

To resolve a resource conflict, an agent needs to determine the relative priority of competing tasks, assign control of the resource to the winner, and decide what to do with the loser. The latter issue differentiates strategies for resolving the conflict. There are at least three basic strategies (cf. (Schneider and Detweiler, 1988)).

Shedding: eliminate low importance tasks
Delaying/Interrupting: force temporal separation between conflicting tasks
Circumventing: select methods for carrying out tasks that use different resources

Shedding involves neglecting or explicitly foregoing a task. This strategy is appropriate when demand for a resource exceeds availability. For the class of resources we are presently concerned with, those which become blocked when assigned to a task but are not depleted by use, demand is a function of task duration and task temporal constraints. For example, a task can be characterized as requiring the gaze resource for 15 seconds and having a completion deadline 20 seconds hence.

In APEX, steps are assumed to be concurrently executable unless otherwise specified. The **waitfor** clause is used to indicate ordering constraints. The general form of a waitfor clause is *(waitfor <eventform>*)* where eventforms can be either a procedure step-identifier or any parenthesized expression. Tasks created with waitfor conditions start in a pending state and become enabled only when all the events specified in the waitfor clause have occurred. Thus, tasks created by steps s1 and s2 begin enabled and may be carried out concurrently. Tasks arising from the remaining steps begin in a pending state.

```
(procedure
    (index (turn-on-headlights)
    (step s1 (clear-hand left-hand))
    (step s2 (determine-loc headlight-ctl => ?loc)
    (step s3 (grasp knob left-hand ?loc)
         (waitfor ?s1 ?s2))
    (step s4 (pull knob left-hand 0.4) (waitfor ?s3))
    (step s5 (ungrasp left-hand) (waitfor ?s4))
    (step s6 (terminate) (waitfor ?s5)))
```

Figure 1 Example PDL procedure

Events arise primarily from two sources. First, perceptual resources (e.g. vision) produce events such as *(color object-17 green)* to represent new or updated observations. Second, the sketchy planner produces events in certain cases, such as when a task is interrupted or following execution of an enabled **terminate** task (e.g. step s6 above). A terminate task ends execution of a specified task and generates an event of the form *(terminated <task> <outcome>)*; by default, <task> is the terminate task's parent and <outcome> is 'success.' Since termination events are often used as the basis of task ordering, waitfor clauses can specify such events using the task's step identifier as an abbreviation – for example, *(waitfor (terminated ?s4 success)) = (waitfor ?s4).*

Detecting Conflicts

The problem of detecting conflicts can be considered in two parts: (1) determining which tasks should be checked for conflict and when; and (2) determining whether a conflict exist between specified tasks. APEX handles the former question by checking for conflict between task pairs in two cases. First, when a task's non-resource preconditions (waitfor conditions) become satisfied, it is checked against ongoing tasks. If no conflict exists, its state is set to *ongoing* and the task is executed. Second, when a task has been delayed or interrupted to make resources available to a higher priority task, it is given a new opportunity to execute once the needed resource(s) become available – i.e. when the currently controlling task terminates or becomes suspended. The delayed task is then checked for conflicts against all other pending tasks.

Determining whether two tasks conflict requires only knowing which resources each requires. However, it is important to distinguish between two senses in which a task may require a resource. A task requires *direct control* in order to elicit primitive actions from the resource. For example, checking the fuel gauge in an automobile requires direct control of gaze. Relatively long-lasting and abstract tasks require *indirect control*, meaning that they are likely to be decomposed into subtasks that need direct control. For example, the task of driving an automobile requires gaze in the sense that many of driving's constituent subtasks involve directing visual attention.

Indirect control requirements are an important predictor of direct task conflicts. For example, driving and visually searching for a fallen object both require indirect control over the gaze resource, making it likely that their respective subtasks will conflict directly. Anticipated conflicts of this sort should be resolved just like direct conflicts – i.e. by shedding, delaying, or circumventing.

Resources requirements for a task are undetermined until a procedure is selected to carry it out. For instance, the task of searching for a fallen object will require gaze if performed visually, or a hand resource if carried out by grope-and-feel. PDL denotes resource requirements for a procedure using the PROFILE clause. For instance, the following clause should be added to the turn-on-headlights procedure described above:

```
(profile (left-hand 8 10))
```

The general form of a profile clause is
(profile (<resource> <duration> <continuity>))*. The <resource> parameter specifies a resource defined in the resource architecture – e.g. gaze, left-hand, memory-retrieval; <duration> denotes how long the task is likely to need the resource; and <continuity> specifies how long an interrupting task has to be before it constitutes a *significant interruption*. Tasks requiring the resource for an interval less than the specified continuity requirement are not considered significant in the sense that they do not create a resource conflict and do not invoke interruption-handling activities (as described further on).

For example, the task of driving a car should not be interrupted in order to look for restaurant signs near the side of the road, even though both tasks need to control gaze. In contrast, the task of finding a good route on a

In low workload, the situation is reversed. With enough time to do all current tasks, importance may be irrelevant. The agent must only ensure that deadlines associated with each task are met. In these conditions, urgency should dominate the computation of task priority. The tradeoff between urgency and importance can be represented by the following equation:

$$\text{priority}_b = S*I_b + (S_{max}-S)U_b$$

S is subjective workload (a heuristic approximation of actual workload); I_b and U_b represent importance and urgency for a specified basis (b). To determine a task's priority, APEX first computes a priority value for each basis, and then selects the maximum of these values.

Interruption Issues

A task acquires control of a resource in either of two ways. First, the resource becomes freely available when its current controller terminates. In this case, all tasks whose execution awaits control of the freed up resource are given current priority values; control is assigned to whichever task has the highest priority. Second, a higher priority task can seize a resource from its current controller, interrupting it and forcing it into a suspended state.

A suspended task recovers control of needed resources when it once again becomes the highest priority competitor for those resources. In this respect, such tasks are equivalent to pending tasks which have not yet begun. However, a suspended task may have ongoing subtasks which may be affected by the interruption. Two effects occur automatically: (1) subtasks no longer inherit priority from the suspended ancestor and (2) each subtask is reprioritized, possibly causing it to become interrupted. Other effects are procedure-specific and must be specified explicitly. PDL includes several primitives steps useful for this purpose, including RESET and TERMINATE.

```
(step s4 (turn-on-headlights))
(step s5 (reset) (waitfor (suspended ?s4))
```

For example, step s5 causes a turn-on-headlight task to terminate and restart if it ever becomes suspended. This behavior makes sense because interrupting the task is likely to undo progress made towards successful completion. For example, the driver may have gotten as far as moving the left hand towards the control knob at the time of suspension, after which the hand will likely be moved to some other location before the task is resumed.

Robustness against interruption

Discussions of planning and plan execution often consider the need to make tasks robust against failure. For example, the task of starting an automobile ignition might fail. A robust procedure for this task would incorporate knowledge that, in certain situations, repeating the turn-key step is a useful response following initial failure. The possibility that a task might be interrupted raises issues similar to those associated with task failure, and similarly requires specialized knowledge to make a task robust. The problem of coping with interruption can be divided into three parts: wind-down activities to be carried out as interruption occurs, suspension-time activities, and wind-up activities that take place when a task resumes.

It is not always safe or desirable to immediately transfer control of a resource from its current controller to the task that caused the interruption. For example, a task to read information off a map competes for resources with and may interrupt a driving task. To avoid a likely accident following abrupt interruption of the driving task, the agent should carry out a wind-down procedure (Gat, 1992) that includes steps to, e.g., pull over to the side of the road. The following step within the driving procedure achieves this behavior.

```
(step s15 (pull-over)
  (waitfor (suspended ?self))
  (priority (avoid-accident) (importance 10)
       (urgency 10)))
```

Procedures may prescribe additional wind-down behaviors meant to (1) facilitate timely, cheap, and successful resumption, and (2) stabilize task preconditions and progress – i.e. make it more likely that portions of the task that have already been accomplished will remain in their current state until the task is resumed. All such actions can be made to trigger at suspension-time using the waitfor eventform *(suspended ?self)*.

In some cases, suspending one task should enable others meant to be carried out during the interruption interval. Typically, these will be either monitoring and maintenance tasks meant, like wind-down tasks, to insure timely resumption and maintain the stability of the suspended task preconditions and progress. Windup activities are carried out before a task regains control of resources and are used primarily to facilitate resuming after interruption. Typically, windup procedures will include steps for assessing and "repairing" the situation at resume-time – especially including progress reversals and violated preconditions. For example, a windup activity following a driving interruption and subsequent pull-over behavior

Other prospective refinements to current mechanisms include allowing a basis to be suppressed if its associated factor is irrelevant in the current context, and allowing prioritization decisions to be made between compatible task groups rather than between pairs of tasks. The latter ability is important because the relative priority of two tasks is not always sufficient to determine which should be executed. For example: tasks A and B compete for resource X while A and C compete for Y. Since A blocks both B and C, their combined priority should be considered in deciding whether to give resources to A.

Perhaps the greatest challenge in extending the present approach will be to incorporate deliberative mechanisms needed to optimize multitasking performance and handle complex task interactions. The current approach manages multiple tasks using a heuristic method that, consistent with the sketchy planning framework in which it is embedded, assumes that little time will be available to reason carefully about task schedules. Deliberative mechanisms would complement this approach by allowing the agent to manage tasks more effectively when more time is available.

Acknowledgements

Thanks to Jim Johnston, Roger Remington, and Michael Shafto for their interest and support, and to Barney Pell for comments on a previous draft of this paper.

References

Cohen, P.R., Greenberg, M.L., Hart, D., and Howe, A.E. 1989. An Introduction to Phoenix, the EKSL Fire-Fighting System. EKSL Technical Report,. Department of Computer and Informational Science. University of Massachusetts, Amherst.

Firby, R.J. 1989. Adaptive Execution in Complex Dynamic worlds. Ph.D. thesis, Yale University.

Freed, M. & Remington, R.W. 1997. Managing Decision Resources in Plan Execution. In Proceedings of the Fifteenth Joint Conference on Artificial Intelligence, Nagoya, Japan.

Freed, M. 1998. Simulating human performance in complex, dynamic environments. Ph.D. thesis, Northwestern University.

Gat, Erann. 1992. Integrating planning and reacting in heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of 1992 National Conference on Artificial Intelligence.*

Georgeff, M and Lansky, A. 1987. Reactive Resoning and Planning: An Experiment with a Mobile Robot. *Proceedings of 1987 National Conference on Artificial Intelligence.*

Hayes-Roth, B. 1995. An architecture for adaptive intelligent systems. Artificial Intelligence, 72, 329-365.

Pell, B., Bernard, D.E., Chien, S.A.., Gat, E., Muscettola, N., Nayak, P.P., Wagner, M., and Williuams, B.C. 1997. An autonomous agent spacecraft prototype. *Proceedings of the First International Conference on Autonomous Agents*, ACM Press.

Schneider, W. and Detweiler, M. 1988. The Role of Practice in Dual-Task Performance: Toward Workload Modeling in a Connectionist/Control Architecture. *Human Factors*, 30(5): 539-566.

Simmons, R. 1994. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation.* 10(1).