Task Networks for Controlling Continuous Processes

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

Abstract

This paper describes an extension to the RAP system task-net semantics and representation language to enable the effective control of continuous processes. The representation addresses the problems of synchronizing plan expansion with events in the world, coping with multiple, non-deterministic task outcomes, and the description of a simple form of *clean-up* task.

It is also pointed out that success and failure need no special place in a task network representation. Success and failure are really messages about the execution system's knowledge and do not explicitly define that system's flow of control.

Introduction

Recently, AI researchers have proposed several different mechanisms for programming robots reactively. These include collections of behaviors (Brooks 1986), schemas (Arkin 1987), routines (Gat 1992), and reflexes (Payton 1986). Many details differ between these proposals, particularly in the area of philosophical commitment, but they share the common idea that the actual behavior of the robot at any given moment is the result of a set of interacting processes acting on input from the environment. Thus, the behavior of the robot (i.e., its apparent immediate goal) can be changed by changing the set of active processes. This idea has been discussed by several authors and it allows some aspects of robot control to be described in terms of concurrent processes while other aspects are described in terms of discrete, symbolic steps that enable and disable those processes (Firby 1992; Slack 1992; Gat 1991; Firby & Swain 1991; Payton 1990).

Given that a desired robot action is obtained by enabling an appropriate set of processes, there is still the problem of telling when a particular goal has been attained or when a situation has arisen that prevents that goal from being attained. Often the processes themselves can detect important states of the world, particularly those in which the process is not functioning properly. It will also often be necessary to use processes to detect transient conditions reliably. We assume that when processes detect various conditions, either good or bad, they will generate asynchronous signals. Depending on the programming model used, signals might be generated by processes directly, they might correspond to particular values appearing on wires connecting behaviors, or they might occur when a particular process is invoked. In all cases, however, signals will be fairly low-level messages from some process that does not always know the goals it will be used to achieve.

*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 Animate Agent Architecture attempts to integrate these ideas using the interface between reactive execution and continuous control illustrated in Figure 1. The RAP system takes tasks and refines them into commands to enable appropriate sensing and action processes for the situation encountered at run time. Typically, processes will be enabled in sets that correspond to the notion of discrete "primitive steps" that will reliably carry out an action in the world over some period of time. Slack has dubbed such collections of processes "reactive skills" (Slack 1992). The RAP system produces goal-directed behavior using this idea by refining abstract plan steps into a sequence of different configurations for a process-based control system.

Controlling Processes with the RAP System

Once a set of processes has been started up, the RAP system relies on signals to tell it when the desired activity is complete and how it came out. Since the reason for invoking a set of processes is not known to the processes themselves, the RAP system must interpret signals in context. The same signals might mean different things in different plans. For example, a process for approaching a given target might be used to move up to a fixed object in the world or it might be used to follow a moving target around. A signal saying that the target has been reached means the task is complete when approaching but it means the object is too close when following.

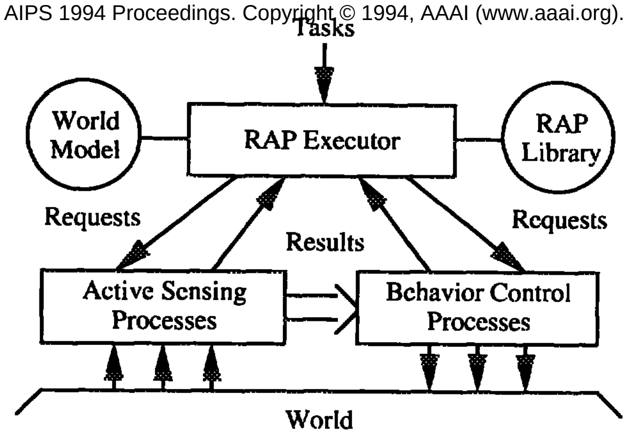


Figure 1: The Animate Agent Architecture

Thus, a RAP task description must:

1. Allow concurrent threads of execution so that multiple control processes can be started up together.
2. Represent when to proceed to the next subtask in a method given that the task must wait for certain signals to do so.
3. Describe methods for a task that allow different next steps when different signals are received.

This paper discusses these issues and presents a new task method representation language for the RAP system.

The RAP System

The RAP system is designed for the reactive execution of symbolic plans. A plan is assumed to include goals, or tasks, at a variety of different levels of abstraction and the RAP system attempts to carry out each task in turn using different methods in different situations and dealing with common problems and simple interruptions.

In the RAP system a task is described by a RAP which is effectively a context sensitive program for carrying out the task. The RAP can also be thought of as describing a variety of plans for achieving the task in different situation. For the purposes of this paper, the important aspects of a RAP task description are the SUCCEED and METHOD sections.

For example, the following RAP describes how to pick something up in the simulated delivery world used in initial RAP system development (Firby 1989; Firby & Hanks 1987):

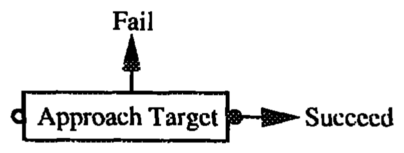
```
(define-rap (arm-pickup ?arm ?thing)
  (succeed (ARM-HOLDING ?arm ?thing))
  (method
    (context (not (TOOL-NEEDED ?thing ?tool true)))
    (task-net
      (t1 (arm-move-to ?arm ?thing) (for t2))
      (t2 (arm-grasp-thing ?arm ?thing))))
  (method
    (context (TOOL-NEEDED ?thing ?tool true))
    (task-net
      (t1 (arm-pickup  ?arm ?tool) (for t2))
      (t2 (arm-move-to ?arm ?thing)(for t3))
      (t3 (arm-grasp-thing ?arm ?thing)))))
```

This RAP has two methods for achieving the goal. The SUCCEED clause is a predicate checked against memory to see if the overall task is complete. Each METHOD specifies a plan, or TASK-NET, for achieving the SUCCEED condition in a given CONTEXT. Like the SUCCEED clause, each CONTEXT is a predicate to be checked in memory. This paper is about writing TASK-NETs that link subtasks, process invocations, and signal interpretation into a coherent plans.

The RAP system (Firby 1987; 1989) carries out tasks using the following algorithm. First, a task is selected for execution and if it represents a primitive action, it is executed directly, otherwise its corresponding RAP is looked up in the library. Next, that RAP's check for success is used as a query to the situation description and, if satisfied, the task is considered complete and the next task can be run. However, if the task has not yet been satisfied, its method-applicability tests are checked and one of the methods with a satisfied test is selected. Finally, the subtasks of the chosen method are queued for execution in place of the task being executed, and that task is suspended until the chosen method is complete. When all subtasks in the method have been executed, the task is reactivated and its completion test is checked again. If all went well the completion condition will now be satisfied and execution can proceed to the next task. If not, method selection is repeated and another method is attempted.

Task/Goal Semantics

An important aspect of representing and executing a plan is the meaning of a subgoal or subtask. The RAP system was originally written assuming that RAP method subtasks can be treated as atomic. From the point of view of the method using a subtask, it will either succeed or fail, and it will not complete until that success or failure is known.



```
(task-net
  (t1 (approach-target ?target)))
```

Figure 2: A Symbolic, Discrete Task

For example, the task network shown in Figure 2 contains one subtask and once that subtask is spawned by the interpreter, further processing of the method will stop until the subtask completes. It is assumed that the subtask will either succeed, in which case the interpreter should continue processing the task network after the task (from the black dot in the figure) or it will fail, in which case the method as a whole should fail and all of its subtasks should be terminated (in this example there aren't any).

This representation and semantics for a subtask (or method, or *plan*), assumes that it is appropriate to execute the next step in a method as soon as a subtask completes. This assumption is pervasive in the literature and it makes perfect sense when subtasks are truly atomic. In fact, one of the motivating ideas behind the RAP system is to try and make this property hold by working on each task until it is known to have succeeded.

This semantics also embodies another, more subtle assumption. It assumes that a subtask actually has a well-defined finish and will know when it has succeeded or failed. This assumption also makes sense when actions are atomic. In fact, this assumption is what *makes* actions primitive or atomic.

Continuous Tasks and Signals

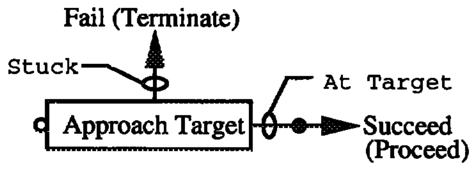
Unfortunately, given the low-level robot control system used in the Animate Agent Architecture, neither of these assumptions holds. Goals are achieved by sets of processes that must be enabled independently and detecting goal completion depends on the appropriate interpretation of signals from those processes. Thus, RAP methods must explicitly define which signals mean a subtask has succeeded and which signals mean it has failed.

When robot activity is controlled by enabling and disabling sets of processes, time must pass while the activity is underway. If the RAP system does the enabling and disabling explicitly, methods must have a way to let time pass and synchronize further task expansion with process progress.

We have adopted a task net annotation that tells the interpreter to WAIT-FOR a given signal before proceeding to the next subtask in a method (this idea is closely related to McDermott's notion of blocking a task thread while waiting for a fluent (McDermott 1992)). For example, a method to approach a fixed target might look like that shown in Figure 3.

This method executes the subtask **approach-target** and then waits for either an **(at-target)** signal or a **(stuck)** signal. If the **(at-target)** signal is received first, then this subtask succeeds and if **(stuck)** arrives first, this subtask fails and other subtasks in the method are terminated.

The diagram shows WAIT-FOR clauses as gating conditions for letting the interpreter proceed with executing the method. The black dots correspond to the more traditional interpretation of when the subtask has completed.

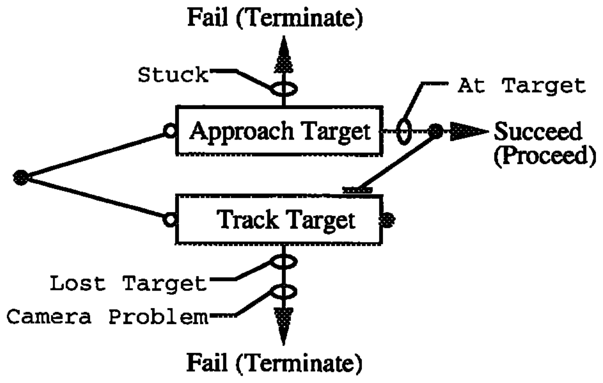


```
(task-net
  (t1 (approach-target ?target)
      (wait-for (at-target) :proceed)
      (wait-for (stuck) :terminate)))
```

Figure 3: Waiting for A Signal to Proceed

Concurrent Tasks

The RAP task net representation has always supported non-linear methods with parallel threads of execution. This ability to support concurrent tasks is critical when the RAP system is being used to enable and disable processes in concurrent sets. For example, consider the method shown in Figure 4.



```
(task-net
  (t1 (approach-target ?target)
      (wait-for (at-target) :proceed)
      (wait-for (stuck) :terminate))
  (t2 (track-target ?target)
      (wait-for (lost-target) :terminate)
      (wait-for (camera-problem) :terminate)
      (until-end t1)))
```

Figure 4: A Simple Concurrent Task Net

Both of the primitive processes **approach-target** and **track-target** must be active to get the robot to visually servo to a selected fixed object. The task net does not order these tasks and the interpreter follows two threads of execution. Choosing one thread to follow first, the interpreter starts the indicated process and then blocks that thread until receipt of one of the indicated signals. While that thread is blocked, the interpreter follows the other thread, enabling the other process and blocking that thread while waiting for one of its signals. The **until-end** annotation in the task net tells the interpreter that subtask **t2** should be terminated whenever subtask **t1** completes. The only way this method can complete successfully is to receive the **(at-target)** signal from process **approach-target**.[1]

The critical step in concurrent RAP task processing when primitive actions start processes rather than achieve goals is not getting concurrent task expansion to work, but rather, stopping expansion in the right places to synchronize further processing with the actual progress of the task in the real world. This synchronization is achieved using WAIT-FOR clauses.

The ability of a plan representation formalism to explicitly allow both concurrency and synchronizing execution with events in the world is crucial if the plans are to be used to control a real-time system made up of composable, concurrent processes.

Success and Failure: Red Herrings

Another deeply ingrained assumption in task representation is that a given task will have only two outcomes: success or failure. The original RAP system again tried to enforce this assumption by giving each task a SUCCESS clause that must be satisfied before it stops trying various methods to achieve its goal. However, as has already been illustrated in Section , it isn't very meaningful to say that a task to enable a control process succeeds. Such tasks will invariably succeed by starting the process in motion. What matters are the signals that are generated by the process while it is running. Some signals will mean success and some will mean failure. Hence, WAIT-FOR annotations include the appropriate outcome to use for the task as a whole when the signal is received.

However, grouping the initiating of a process and its termination signals (both good and bad) into a logical unit using WAIT-FOR clauses is too limiting. In fact, the whole idea of success and failure is too limiting.

The Problem of Cleanup Tasks

Consider the problem of local cleanup tasks. When an activity requires enabling a variety of processes and then waiting for some event in the world, forcing the method defining the activity to either succeed or fail on the event makes it very difficult to encode steps in the method to "clean up" the situation when a failure occurs. For example, let's assume that the **track-target** process used previously requires that the robot's camera be turned on first. It is a simple matter to include a task that comes before the **track-target** task to turn the camera on, but where should the corresponding task to turn the camera off go? Placing it after receipt of the **(at-target)** signal makes sense if everything works out. However, if the target is lost and the **(lost-target)** signal is received, the method will fail and terminate before ever reaching the task to turn the camera off.

Forcing the interpretation of signals as success and failure prevents coding methods that include actions to be taken regardless of what happens. In the RAP system, and most other planning systems in the literature, when a subtask fails, all other subtasks in the same method are assumed no longer valid and terminated.

The problem of cleanup tasks is well known in the literature (Gat 1991; McDermott 1991) and the natural inclination is to start trying to represent explicit cleanup or failure recovery plans. However, a closer look at the problem shows that the whole idea of success, failure, and recovery is a red-herring. The real problem is that tasks have multiple outcomes. The **track-target** task doesn't succeed or fail, it might continue to track, it might lose the target, it might suggest a better tracking method, the camera might fail, or a whole host of other possibilities. Each of these outcomes may require a different interpretation and a different course of action.[2]

Non-Deterministic Task Networks

What we really need is a task net representation that explicitly recognizes when subtasks have multiple outcomes and allows a different thread to be followed for each one. Given such a representation, success and failure are possible outcomes, like signals, that can be used to change the flow of control. Success and failure in themselves mean nothing.

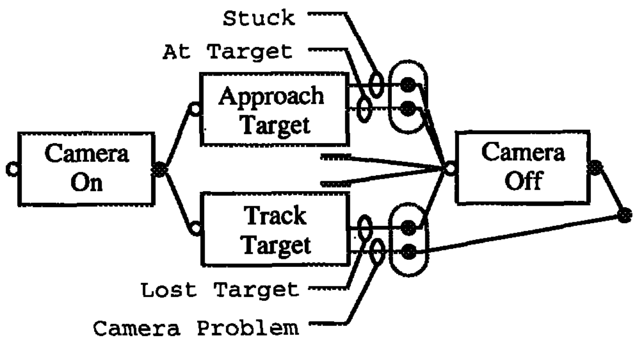
This idea is incorporated into RAP task nets using the WAIT-FOR clause introduced above. As already hinted at in the examples, the third argument can be **:proceed** or **:terminate** or the tag of the next task to execute when the signal occurs. Consider the method defined by the task net shown in Figure 5.

In this method, each WAIT-FOR clause points to the next subtask to execute upon receipt of the appropriate signal. Notice that **:success** is treated as a signal instead of a result. The RAP system also uses the FOR clause to specify the next step in a plan which, given the semantics of success as a signal, is really just a short form for **(wait-for :success *task*)**. The UNTIL-START annotation is similar to UNTIL-END, specifying that both task **t1** and **t2** should terminate when task **t3** starts. Thus, if either **t1** or **t2** completes and passes control to **t3**, the other will stop.[3]

[1] Don't worry about the arguments to approach-target and **track-target**. In reality processes like these must know how to exchange target information in real time below the level of the RAP system.

[2] This mechanism only accounts for a very simple type of cleanup task. The problem of cleaning up a situation before dealing with an interruption is much more difficult. One approach is to use POLICY and PROTECTION triggered tasks as suggested by McDermott (McDermott 1992) but the general problem requires making complex tradeoffs between courses of action at runtime.

[3] Don't worry about the fact that not turning off the camera but terminating the method on a



```
(task-net
  (t0 (camera-on) (wait-for :success t1) (for t2))
  (t1 (approach-target ?target)
      (wait-for (at-target) t3)
      (wait-for (stuck) t3)
      (until-start t3))
  (t2 (track-target ?target)
      (wait-for (lost-target) t3)
      (wait-for (camera-problem) :terminate)
      (until-start t3))
  (t3 (camera-off)))
```

Figure 5: A Complex Task Net

It is also important to note that there is no longer any notion of failure. Should the **(camera-problem)** signal be received while task **t2** is active, the method as a whole is terminated; terminating all of its active subtasks. The previous semantics given to task failure is subsumed by an explicit directive to terminate the method. Of course, it is possible to use **:fail** as a signal to be caught from the subtask so that if it explicitly fails, control can be passed to an appropriate followup subtask. By default, all subtasks are assumed to have an implicit **(wait-for :fail :terminate)** annotation unless an explicit WAIT-FOR failure is included. This default assumption leaves previous RAP task nets with their original semantics.

Related Work

The Reactive Plan Language proposed by Mcdermott is particularly relevant to the issues addressed in this paper (McDermott 1991). The RPL system includes mechanisms for making tasks wait until particular signals arrive and for multiple threads of plan execution. A goal of that work is to define a language and interpreter that allows plans for arbitrary processes so that the same language used to describe tasks and plans can be used to describe low-level feedback loops as well.

With the addition of signals and multiple outcomes, the similarity between RAP task networks and finite state automata becomes more compelling. RAP task networks don't explicitly describe finite state automata because tasks don't correspond directly to states of execution but the similarity suggests connections to control theory and work that attempts to bridge the gap between AI and control theory. In particular, subsumption based robots use behaviors described by finite state automata (Brooks 1986) and the CIRCA system reasons about plans as finite state automata to construct provably safe control loops (Musliner, Durfee, & Shin 1993).

Discrete event system theory is directed toward building and understanding control plans for systems that can be described as finite state automata (Ramadge & Wonham 1987). These ideas have been used to build simple control systems for vision-based robot navigation problems (Kosecka & Bajcsy 1993). The Animate Agent project uses an underlying control system that can be reconfigured into different states, and RAP task networks are plans for sequencing those states in response to changing goals and events. Thus, in some sense, RAP task networks are control plans for a discrete event system. Currently, RAP task networks are intended to cover a broad range of behavior but do not attempt to guarantee controllability or stability over all task goals. We will be exploring tighter connections between the RAP system and discrete event control theory in the future.

Conclusions

The symbolic planning notion of primitive, atomic actions cannot readily be used to control the enabling and disabling of processes that must act together over time to achieve goals in the world. When primitive actions start processes in motion and the same processes can be used in different combinations to achieve different goals, the planning system can no longer assume that primitive actions will have their own well-defined end points.

This paper proposes an extension to the RAP system task-net representation and semantics to enable the effective control of such continuous processes. In particular:

- The RAP task net's ability to represent concurrent subtasks can be used to enable many subtasks/processes simultaneously. This allows a method to individually enable all of the processes required to carry out a particular activity in the world.
- RAP task nets are extended to include WAIT-FOR clauses between plan steps. A WAIT-FOR clause specifies a condition that must be true in the world before the interpreter can move on to the next task in the method. This allows the expansion/execution of subtasks to be synchronized with events in the world. Synchronization was previously assumed to be inherent in the success of primitive actions.
- The WAIT-FOR clause is also used to allow different threads of execution to proceed from different task outcomes. Effectively, each WAIT-FOR represents a

(camera-problem) signal makes no sense. This example is intended simply to illustrate the ideas.

different possible task outcome and can specify the next subtask to execute after the WAIT-FORs condition becomes true.

This task network representation has been implemented and is being used to control a real robot doing vision-based navigation tasks at the University of Chicago.

This paper also argues that success and failure have no special place in a task network representation. Success and failure are interpretations placed on certain task outcomes based on what is desired. In fact, the RAP system already has little use for explicit failure. Whether a RAP method succeeds or fails, the task using the method does not succeed unless its explicit SUCCESS condition is true. Method success is simply method completion and method failure means that all subtasks in the method should be terminated. With the explicit representation of multiple outcomes for a subtask, and the ability for an outcome to explicitly terminate a method, failure and success are no longer necessary.

The notion of a failure still exists in the RAP system but it means exclusively that the system does not *know how* to achieve the task. The only way a task can fail is if all of its methods have been tried repeatedly and its success condition has not been met: either no method applies in the current situation or applicable methods appear to have no useful effect on the world. Thus, failure is a signal about the planning knowledge of the system and not about the execution result of a task.

Acknowledgements

Many of these ideas were made clear during very interesting discussions with Marc Slack and Tim Converse.

References

Arkin, R. C. 1987. Motor schema based navigation for a mobile robot: An approach to programming by behavior. In *International Conference on Robotics and Automation*. Raleigh, NC: IEEE.

Brooks, R. A. 1986. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation* RA-2(1).

Firby, R. J., and Hanks, S. 1987. A simulator for mobile robot planning. In *Knowledge-Based Planning Workshop*. Austin, TX: DARPA.

Firby, R. J., and Swain, M. J. 1991. Flexible task-specific control using active vision. In *Sensor Fusion IV: Control Paradigms and Data Structures*. Boston, MA: SPIE.

Firby, R. J. 1987. An investigation into reactive planning in complex domains. In *Sixth National Conference on Artificial Intelligence*. Seattle, WA: AAAI.

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

Firby, R. J. 1992. Building symbolic primitives with continuous control routines. In *First International Conference on AI Planning Systems*.

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

Gat, E. 1992. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Tenth National Conference on Artificial Intelligence*. San Jose, CA: AAAI.

Kosecka, J., and Bajcsy, R. 1993. Cooperation of visually guided behaviors. In *Fourth International Conference on Computer Vision*. Berlin, Germany: IEEE.

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.

Musliner, D. J.; Durfee, E. H.; and Shin, K. G. 1993. Circa: A cooperative intelligent real-time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics* 23(6).

Payton, D. 1986. An architecture for reflexive autonomous vehicle control. In *International Conference on Robotics and Automation*. San Francisco, CA: IEEE.

Payton, D. 1990. Exploiting plans as resources for action. In *Workshop on Innovative Approaches to Planning, Scheduling, and Control*. San Diego, CA: DARPA.

Ramadge, P. J., and Wonham, W. M. 1987. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization* 25(1).

Slack, M. 1992. Sequencing formally defined reactions for robotic activity: Integrating raps and gapps. In *Vol. 1828 Sensor Fusion V: Simple Sensing for Complex Action*. Boston, MA: SPIE.