From: AAAI Technical Report FS-96-01. Compilation copyright © 1996, AAAI (www.aaai.org). All rights reserved.

Executing Structured Reactive Plans*

Michael Beetz
Institut für Informatik III,
Universität Bonn, Römerstr. 164,
D-53117 Bonn, Germany,
beetz@cs.uni-bonn.de

Drew McDermott
Yale University,
Department of Computer Science,
P.O. Box 208285, Yale Station,
New Haven, CT 06520-8285, USA,
mcdermott@cs.yale.edu

Introduction

We consider plan execution for an autonomous robot to be a computational process that performs the following task: given (1) a plan that has been designed to accomplish some jobs, (2) a stream of sensory data, and (3) a stream of feedback signals from the robot control processes, cause the robot to exhibit a behavior that accomplishes its jobs as specified by the plan. Plan execution would be trivial if the plan specified all aspects of the intended behavior for all possible streams of sensory input and feedback signals. Such detailed plans, however, are often extremely complex and impossible to synthesize automatically.

An elegant solution to this problem are robot control systems with planning systems that synthesize behaviors specified in terms of discrete, atomic plan steps with ordering constraints. The behavior specifications produced by these planning systems are abstract and sketchy and do not specify how the robot is to react to asynchronous occurrences of events, arrival of sensor data, to synchronize its concurrent actions, and to recover from execution failures (Fir89). In this kind of architecture, a plan execution module has to produce competent behaviors from simple (sketchy, abstract) behavior specifications and is responsible for the successful execution of the individual steps and checking the progress of plan execution in all possible situations (Fir89; HF90).

In many applications, however, robots should exhibit very different behaviors when executing similar plans in similar situations. Consider the following two plans for transporting objects: first, the robot is to get the bomb from location l to location l' (to prevent an explosion) and second, the robot is to get the empty box from l to location l' (to store some objects). Both plans are very similar: the robot should go to l, get the respective object and deliver it to l'. If, however, both plans were to be executed in a situation where more than one bomb-like object or empty box are at l the robot should behave very differently: In the first case the robot should bring *all* the bomb-like objects to l'. In the second case *any* empty box will do.

To obtain reactive behaviors that are tailored to the objectives of the plans it seems advantageous that the plan pieces themselves spell out the methods for their robust execution. XFRM is an architecture for controlling a simulated robot acting in a changing and partly unknown environment that does just this (McD90). The architecture contains a planning system (McD92; BM94; Bee96) generating plans that specify how the robot is to respond to sensory input and feedback from its control processes to accomplish its jobs reliably without wasting resources.

Consequently, we don't have much to say about how XFRM's plans are executed — they are simply run. Instead, we will focus on the nature of plans that can specify robust execution strategies in changing and partly unknown environments. We will argue that such plans should specify behavior as the result of concurrent control routines, driven by sensor information. In addition, such plans need mechanisms for signalling and handling execution failures.

The DeliveryWorld

The requirements for executable plans depend on the characteristics of the environments the robots act in, the robots' jobs and their capabilities. For example, if random asynchronous events occur that might interfere with the mission of the robot, the robot has to monitor its surroundings for unforeseen events, react to them immediately as they occur, and resume its mission. Or, if the effectors of the robot are imperfect, the robot might have to sense the effects of its actions and repeat them till they are successful. Or, if its sensors are limited or noisy it might require more robust sensing strategies.

* This work was supported by the Defense Advanced Research Projects Agency, contract number N00014-91-J-1577, administered by the Office of Naval Research.

We will look at a particular control problem: the control of a simulated delivery robot that performs varying jobs in a complex, changing, and partly unknown (simulated) environment. The robot itself has limited and noisy sensing capabilities and its effectors (hands and drive) are imperfect. In the following, we will describe the aspects of the world that challenge robust and efficient plan execution.

Commands and Jobs

The delivery robot is supposed to carry out multiple jobs, which can be changed at any time by adding, deleting, or modifying jobs. The following is a typical set of jobs:

1) get all black blocks from ⟨1,13⟩ to ⟨1,4⟩
2) get all boxes from ⟨1,13⟩ to ⟨4,9⟩
3) paint all pyramids at ⟨1,4⟩ black
4) get the white shiny block from ⟨4,9⟩ to ⟨4,3⟩

The objective of the robot controller is to carry out the given jobs reliably and cost-effectively. The jobs specify conditions on the effects that the robot should have on its environment. If the supervisor commands the robot to deliver an empty box to a specified location, he cares about whether there will be an empty box at the location when the robot is done and how fast the box will get there. An experimenter can assign the robot a score for a given period based on the number of jobs it has performed, the degree to which it has accomplished them, and the time it has spent.

The World

Currently, our simulated robot does errand planning in a simulated world of discrete locations developed by McDermott and briefly described in (McD92). The world is a grid of locations; each location has an infinite number of positions and at each position there is at most one object. Boxes, blocks, and balls with different colors, textures, and sizes, are spread out in the world and can move, appear, disappear, or change their appearance due to exogenous events. Exogenous events occur randomly and asynchronously to the actions of the robot. Additional autonomous robots inhabit the world and perform jobs like cleaning locations or delivering and modifying objects. The jobs of different robots might interfere and require robot controllers to adjust to these interferences. Time passes at a realistic speed: that is, realistic estimates are used for the time it takes the robot to perform sensing and effector actions.

Interference caused by exogenous events and other robots occur rarely. Thus, the robot can make use of *routine plans*. Routine plans that assume the absence of exogenous events and other robots have a good chance of being successful. However, they must watch out for clues that signal exceptional states and adapt themselves, if necessary, to these exceptional situations. One kind of exceptional state are other robots acting in the neighborhood of the robot.

The Robot

XFRM controls a robot with two hands and two cameras. By using a camera the robot can look for objects at its current location, track objects in the camera view, or examine a tracked object. Sensor programs look for objects matching a given description and return a *designator* describing each matching object and its location. Since the robot's cameras are imperfect, designators will be inaccurate and the robot may overlook or hallucinate objects.

To change its environment, the robot can reach out a hand to the positions of tracked objects, grasp them, and thereby pick them up. While grasping or carrying objects, objects might slip out of the robot's hands or might be stuck when the robot opens a hand. At any time the robot can sense and affect only its own location.

While acting in its environment, the robot stores and updates designators and uses them to find and manipulate the objects they describe. The known designators constitute the robot's model of the current state of its environment. This model is necessarily incomplete because it only contains designators for already perceived objects. It may even be wrong since (a) the designators are produced by noisy sensors and (b) exogenous events might have changed the world without the robot noticing.

Specifying Routine Behavior

The routine plans of the delivery robot are specified as structured reactive plans, plans that specify how the robot is to respond to sensory data and feedback to accomplish its jobs. Structured reactive plans are collections of concurrent control routines that specify routine activities and can adapt themselves to non-standard situations by means of planning. We call the controllers reactive because they can respond immediately to asynchronous events and manage concurrent control processes. We call them structured because they use expressive control abstractions to structure complex activities. Structured reactive controllers execute three kinds of control processes: routine activities that handle standard jobs in standard situations, monitoring processes that detect non-standard situations, and planning processes that adapt, if necessary, routine activities to non-standard situations.

Structured reactive plans are implemented in RPL

(McD91). RPL programs look very much like LISP programs that make use of control abstractions typical of structured concurrent programming languages. Such abstractions include those for sequencing, concurrent execution, conditionals, loops, assignments of values to program variables, and subroutine calls. Several high-level concepts (such as interrupts and monitors) are provided and used to synchronize parallel actions and make plans reactive and robust.

In this section we describe three important aspects in the design of routine plans that enable the delivery robot to exhibit robust and efficient behavior:

1. the specification of robot behavior as the result of concurrent control routines;

2. the connections between control routines and sensors that make routines sensor-guided; and

3. the methods for signalling, responding to, and recovering from, execution failures.

Plan Steps vs. Control Processes

To explain why we want the plans controlling the robot in the DELIVERYWORLD to be collections of concurrent control processes rather than sets of discrete action steps, let us first make the assumption that at some level of abstraction the behavior of the robot is sufficiently described by a partially ordered set of discrete steps. This is the level of abstraction at which planning takes place. Reactive and sensor-driven behavior is specified within the steps. Thus, at the "planning layer," the controller for a delivery "deliver object A, which is at location ⟨0,3⟩, to location ⟨4,6⟩" could be specified as a sequence of four steps: (1) (GO ⟨0,3⟩), (2) (PICK-UP A), (3) (GO ⟨4,6⟩), and (4) (PUT-DOWN A).

In the DELIVERYWORLD going over large distances takes considerable time. It is therefore more efficient to carry out steps opportunistically, when the robot happens to be at the location where a given step should be taken. However, steps like "go to a particular location" are too coarsely grained to exploit this kind of opportunism. Therefore, it is necessary to break up these steps into sets of (MOVE *DIR*) steps that bring the robot to the neighboring grid cell in direction *DIR*. In this case, the delivery plan becomes a set of MOVE steps, the PICK-UP step, another set of MOVE steps, and finally the PUT-DOWN step.

Where locations are not blocked, the order in which MOVE steps are carried out does not affect the robot's final position. If locations are blocked, the robot planner computes and carries out optimal navigation plans, which are disjunctions of *partial orders on* MOVE *steps* that bring the robot to its destination without sending it past blocked locations. If, however, the robot does not know in advance which of the locations are blocked, it will compute optimal paths to the destination and execute them until it runs into a blocked location. After running into a blocked location it will stop, plan a new optimal path to circumnavigate the blockage, and start the execution of the new navigation plan. This "stop and replan" strategy is often inefficient: the robot has to wait for the termination of the planning effort before it can take its next step.

Hierarchically structured plans seem to resolve these problems: at a high level of abstraction they ask the robot to "go to ⟨0,3⟩" abstracting away from the details of how to get there, while at a lower level they specify the series of move operations to get there. However, even at the higher level the planning process cannot abstract away from the route the robot will take because the plan might contain pieces for handling opportunities and risks that are triggered by observations that are determined by the route. These plan pieces often have important effects on the intended course of action even at the highest level of abstraction.

Thus, we specify behavior as the result of concurrent control processes with partial orders as a special method for synchronizing the control processes. The specification of robot behavior using concurrent control processes with different priorities is a powerful technique to deal with partly unknown and changing situations (Bro86). Going to a particular location could then be the result of two concurrent control processes: the first causes the robot to go towards the destination until it has arrived there and the second causes it to circumnavigate blocked locations whenever it runs into them (LS86). Opportunities like passing the location where an object has to be painted can be exploited if the "go to destination" process can be interrupted and resumed afterwards.

Another advantage of this approach is that behaviors like exploration, search, and delivery, which are difficult to specify in terms of discrete plan steps, can be implemented as variants of the "go towards" behavior by adding concurrent control processes to the "go" process. For example, if the purpose of going to location ⟨4,6⟩ is to carry an object, it will make sense to activate another process that monitors the force sensor of the robot's hand such that the robot can notice when it loses the object while going to the destination. If the object has been dropped, the robot should immediately stop, look for the object, pick it up, and then continue to ⟨4,6⟩.

Interfacing Continuous Processes

The implementation scheme for structured reactive plans distinguishes between two kinds of control modules: continuous control processes such as grasping and moving and concurrent control routines. The concurrent control routines specify how the robot is to respond to feedback from the continuous control processes and sensor data in order to accomplish its jobs. The purpose of executing control routines is to activate and deactivate control processes, react to asynchronous events, and monitor the execution of the processes. The difficult part in the implementation of such controllers is the specification of the interactions among the concurrent control routines so that they cause the robot to exhibit the intended behavior.

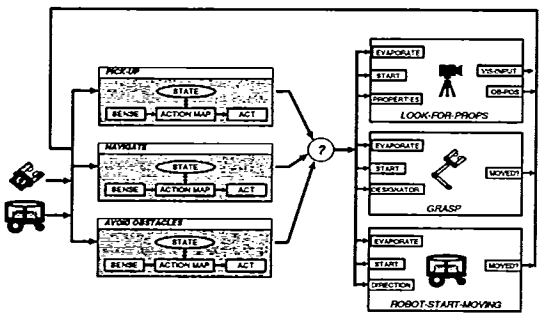

Figure 1: Implementation scheme for routine activities in structured reactive controllers.

To facilitate the interaction between the concurrent control routines and the continuous control processes RPL supports a uniform protocol for the interaction between RPL programs and the control processes. The control abstraction that implements this protocol is called a *behavior module.*

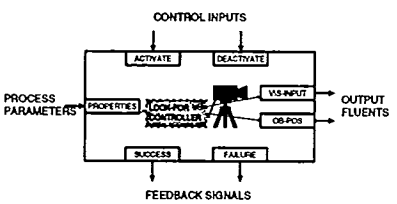

Figure 2: The behavior module LOOK-FOR-PROPS.

Figure 2 shows the behavior module LOOK-FOR-PROPS. LOOK-FOR-PROPS can be activated with a perceptual description of the kinds of objects the robot is looking for (red balls, for instance), and the camera it is supposed to look with. The control process that is performed by the behavior module takes a camera image, searches for red ball-like objects in the image, and stores the position of each red ball in the register OB-POS, a location in the memory.

RPL programs can affect the execution of control processes in two ways only: by activating and parameterizing a control process and by deactivating it. Of course, on deactivation control processes cannot simply stop. The robot must not stop the execution of a process "cross the street" while it is in the middle of the street. RPL provides a construct (EVAP-PROTECT *A B*) that secures the execution of the code piece *B* in the case that *A* evaporates in the midst of its execution.

Control processes recognize the completion of the desired behavior and its success (McD92; Fir94). For instance, the successful completion of a grasp is detected if the force measured by the hand force sensor exceeds a certain threshold. A failed grasp can be detected if fingers have minimal distance to each other while the hand force is still zero. In this case, the control routine signals an execution failure along with a description of the failure. Detecting the termination of actions boils down to discretizing continuous processes (Fir92).

RPL distinguishes between two kinds of communication: one within a thread of control and the other between different threads of control. Within a single thread of control, behavior modules signal an execution failure or the successful completion of the behavior and possibly return values on their termination. For example, the filter of LOOK-FOR-PROPS that searches the image for objects satisfying the given perceptual description, sends a success signal and the set of designators describing the matching objects back to the activating thread of control. Control processes also update global registers to notify other control threads about their execution status. For example, the behavior module LOOK-FOR-PROPS pulses the fluent VIS-INPUT in order to signal that the visual search is completed.

Coordinating Control Processes

Using the concept of behavior modules, we can now specify patterns of control that allow a controller to react to asynchronous events, coordinate concurrent control processes, and use the feedback from control processes to make the behavior of the robot robust and efficient. RPL supports the coordination of concurrent control processes by providing means to declare threads of control to be RPL processes and means for connecting control processes to sensors and effectors. RPL also provides concepts for interprocess communication via shared variables.

To realize the "go to destination" behavior in our example, a programmer might implement two control processes "go towards destination" and "circumnavi-

gate blocked location" that work as follows. The sonars for detecting obstacles should be read frequently and, as soon as an obstacle is detected, the CIRCUMNAVIGATE process should interrupt the GO-TOWARDS process and get the robot around the obstacle. Then the GO-TOWARDS process should continue. To achieve such behaviors a programmer needs to

1. connect RPL control threads to sensors,
2. trigger processes with asynchronous events, and
3. resolve conflicts when active control processes send conflicting commands to the robot's effectors. In our case, the processes "go towards destination" and "circumnavigate blocked locations" might cause a conflict by asking the robot to go into different directions.[1]

Connecting Control Threads to Sensors. To connect RPL control threads to sensors and allow them to react to asynchronous events RPL provides *fluents*, specific types of program variables that can signal changes in their values. Fluents are best understood in conjunction with the RPL statements that respond to changes of fluent values. For instance, the RPL statement (WHENEVER *F B*) is an endless loop that executes *B* whenever the fluent *F* gets the value "true." The statement is implemented such that the control thread interpreting it goes into the state "wait-blocked" as long as *F* is false; waits for the fluent to signal that it has become true; and then resumes the interpretation. Besides WHENEVER, WAIT-FOR and FILTER are other control abstractions that make use of fluents. The RPL expression (WAIT-FOR *F*) blocks a thread of control until *F* becomes true. (FILTER *F B*) executes *B* until the value of *F* becomes false. Fluents can be set and changed by sensors, behavior modules, and the structured reactive plan. Examples of fluents that are set by the robot's sensors are HAND-FORCE-SENSOR, VISUALLY-TRACKED, and LOCATION-AHEAD-IS-BLOCKED. These fluents are used to write pieces of programs that react to changes in the environment and the state of the robot.

Behavior Composition. The question remains as to how the processes GO-TOWARDS and CIRCUMNAVIGATE should be synchronized. The desired behavior is that the robot goes towards its destination. Then, if it is heading towards a blocked location, the circumnavigate process interrupts the GO-TOWARDS process, gets the robot around the blocked locations and returns control until the robot is facing the next blocked location. This is accomplished by the following piece of RPL code:

```
(WITH-POLICY
      (WHENEVER LOCATION-AHEAD-IS-BLOCKED?
            (CIRCUMNAVIGATE-BLOCKED-LOCS DEST))
   (FILTER (NOT (= ROBOT-LOCATION DEST))
      (GO-TOWARDS DEST)))
```

In this example we make use of another RPL construct: (WITH-POLICY *P B*), which means, "execute *B* such that the execution satisfies the policy *P*." In our case, "go towards the location *DEST* until you are there;" is subject to the policy that whenever the robot is heading towards a blocked location go *first* around the blocked location before you continue to go towards the destination. Thus the WITH-POLICY statement can be used to implement the vital behaviors of the robot by turning them into the policies for the statement.

Other control structures that combine plans into more complex activity descriptions are the RPL statements PAR, SEQ, LOOP, and IF. The statement (PAR P_1 ... P_n) specifies that the plans P_1, ..., P_n are to be executed in parallel. (SEQ P_1 ... P_n) asks the robot to execute the plans sequentially. (LOOP *P* UNTIL *C*) executes the plan *P* repeatedly until *C* is satisfied. (IF *C* P_1 P_2) performs P_1 if *C* is true, P_2 otherwise.

Failure Recovery

Besides sensor-driven and synchronized concurrent behavior, the recovery from execution failures is another important aspect of robust robot behavior. Sometimes control processes notice that they have not accomplished their tasks or cannot do so. The process for grasping an object may detect, on completion, that the measured hand force is still low. In this case the continuous control process will send a failure signal using the RPL statement (FAIL :CLASS *C* —*ARGS*—): for example, (FAIL :CLASS UNSUCCESSFUL-GRASP DESIG).

RPL provides several control abstractions that can be used to specify how to react to signalled execution failures. For instance, the statement (N-TIMES *N B* UNTIL *C*) executes *B*, ignoring execution failures, until condition *C* holds but at most *N* times. Or, the statement (TRY-IN-ORDER M_1 ... M_n) executes the methods M_1, ..., M_n in their order until the first one will signal a successful execution.

Using these two constructs we can implement a more robust routine for grasping an initially unknown object. There are different methods. If the object turns out to be a ball smaller than the hand span, the robot can wrap its hand around the ball from above. If the object is a block, the robot can try to grasp it

[1] Of course, conflict resolution and command arbitration are very complex. For instance, we might want the robot go around an obstacle into the direction of the destination (see, for example, navigation in Sonja (Cha90) or navigation using potential field methods (Lat91)). The implementation of such navigation methods might require more complex interactions between the behaviors such as passing the destination as an argument to the circumnavigation process.

by putting its fingers on opposite sides of the block. If the object is a box larger than the hand span, the robot should try a "pinch grasp." The plan asks the robot to grasp the object repeatedly until it has it in its hand; but to give up after failing for the third time. This plan fragment (which has a meaning very similar to that of RAPs proposed by Firby (Fir89)) shows a reactive plan that selects different methods for carrying out a task based on sensory input and tries the methods repeatedly until it is successful or gives up:

```
(LET (SIZE CAT)
  (N-TIMES 3
    (!= SIZE (EXAMINE OB '(SIZE)))
    (!= CAT (EXAMINE OB '(CATEGORY)))
    (TRY-ONE
      ((AND (EQ CAT 'BALL) (< SIZE HAND-SPAN*))
       (WRAP-GRASP OB H 'FROM-ABOVE))
      ((AND (EQ CAT 'BLOCK) (< SIZE HAND-SPAN*))
       (GRIP-GRASP
         OB H (ONE-OF '(FROM-LEFT FROM-RIGHT
                        FROM-FRONT))))
      ((AND (EQ CAT 'BOX) (≥ SIZE HAND-SPAN*))
       (PINCH-GRASP OB H 'FROM-ABOVE)))
  UNTIL (FLUENT-VALUE HAND-FORCE*)))
```

Discussion

Subsequently we will discuss some of the questions raised in the call for participation in the context of the delivery robot in the DELIVERYWORLD:

Which domain characteristics are especially significant? (1) The robot learns information important for choosing the appropriate course of action during plan execution. (2) The DELIVERYWORLD randomly generates asynchronous events; other robots act simultaneously. This requires the robot to monitor aspects of its environment and react based on its perceptions. (3) The robot performs most of the time routine jobs in standard situations; exceptional situations occur less frequently. (4) It is useful, sometimes necessary, to interrupt behaviors and resume them later.

Does the nature of a plan change when one takes into account the possibility of feedback, failure, and recovery? For the delivery robot in the DELIVERYWORLD the answer is yes: a wide spectrum of behavior flaws occur for many different reasons such as ambiguous object descriptions, incomplete and faulty world models, exogenous events, and so on. Avoiding flaws and recovering from them requires context- and purpose-dependent execution strategies. For the DELIVERYWORLD it is not possible to come up with a set of generally applicable mechanisms that carry out most of the jobs of the delivery robot robustly and efficiently. Therefore, the plans themselves have to specify the strategies for their robust and efficient execution. To enable the programmer and planning system to specify these strategies the plan notation has to be more expressive.

How should such a plan be represented to facilitate its execution? Among other concepts a plan notation that facilitates plan execution in the delivery robot should provide concepts for running and synchronizing concurrent control routines, connecting control routines to sensors, and signalling and handling execution failures.

Should plans be sketchy or detailed? The plans used by the delivery robot are detailed but very flexible. They do not specify step for step what the robot is to do but rather specify how the robot is to respond to sensory input to accomplish its jobs.

What are the limitations of your system or theory? Our model of plan execution requires robot planning systems to construct and revise concurrent robot control with very complex structures. We therefore assume that a programmer can specify (ultimately that the robot can learn) a library of routine plans that can accomplish the routine jobs of the robot in standard situations reliably and efficiently. In addition, most of the robot's problem solving has to be routine and the robot has to detect non-standard situations. Finally, we assume that the plans in the library can be implemented such that they enable the robot to anticipate, diagnose, and forestall behavior flaws with simple and fast algorithms. Where these assumptions do not hold, our planning algorithms do not work. For detailed discussions on the construction of, reasoning about, and revision of, structured reactive plans see (McD92; Bee96; BM94; BM92).

References

M. Beetz. *Anticipating and Forestalling Execution Failures in Structured Reactive Plans.* Technical report, Yale University, 1996.

M. Beetz and D. McDermott. Declarative goals in reactive plans. In J. Hendler, editor, *AIPS-92*, pages 3–12, Morgan Kaufmann, 1992.

M. Beetz and D. McDermott. Improving robot plans during their execution. In Kris Hammond, editor, *AIPS-94*, pages 3–12, Morgan Kaufmann, 1994.

R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, pages 14–23, 1986.

D. Chapman. Vision, instruction, and action. Technical Report 1204, 1990.

J. Firby. *Adaptive Execution in Complex Dynamic Worlds.* Technical report 672, Yale University, Department of Computer Science, 1989.

J. Firby. Building symbolic primitives with continuous control routines. In J. Hendler, editor, *AIPS-92*, pages 62–69, Morgan Kaufmann, 1992.

J. Firby. Task networks for controlling continuous processes. In Kris Hammond, editor, *AIPS-94*, pages 49–54, Morgan Kaufmann, 1994.

S. Hanks and R. J. Firby. Issues and architectures for planning and execution. In *Proc. of the Workshop on Innovative Approaches to Planning*, pages 59–70, Scheduling and Control, San Diego, CA, 1990.

J.-C. Latombe. *Robot Motion Planning*. Kluwer, Dordrecht, The Netherlands, 1991.

V. Lumelski and A. Stepanov. Dynamic path planning for a mobile automaton with limited information on the environment. *IEEE Transactions on Automatic Control*, 31(11):1058–1063, 1986.

D. McDermott. Planning reactive behavior: A progress report. In K. Sycara, editor, *Innovative Approaches to Planning, Scheduling and Control*, pages 450–458, San Mateo, CA, 1990. Kaufmann.

D. McDermott. A reactive plan language. Research Report YALEU/DCS/RR-864, Yale University, 1991.

D. McDermott. Transformational planning of reactive behavior. Research Report YALEU/DCS/RR-941, Yale University, 1992.