January 1992

Report No. STAN-CS-92-1412

Toward Agent Programs With Circuit Semantics

by

Nils J. Nilsson

Department of Computer Science

Stanford University

Stanford, California 94305

TOWARD AGENT PROGRAMS WITH CIRCUIT SEMANTICS

Nils J. Nilsson
Robotics Laboratory
Department of Computer Science
Stanford University
Stanford, CA 94305

Tue Jan 21 1992

ABSTRACT

New ideas are presented for computing and organizing actions for autonomous agents in dynamic environments—environments in which the agent's current situation cannot always be accurately discerned and in which the effects of actions cannot always be reliably predicted. The notion of "circuit semantics" for programs based on "teleo-reactive trees" is introduced. Program execution builds a combinational circuit which receives sensory inputs and controls actions. These formalisms embody a high degree of inherent conditionality and thus yield programs that are suitably reactive to their environments. At the same time, the actions computed by the programs are guided by the overall goals of the agent. The paper also speculates about how programs using these ideas could be automatically generated by artificial intelligence planning systems and adapted by learning methods.

I. Control Theory and Computer Science

Designing autonomous agents, such as mobile robots, has been a difficult problem for artificial intelligence because these agents operate in constantly changing environments--environments which can be sensed only imperfectly and affected with only uncertain results. Yet, engineers have long been able to build many automatic devices that function effectively for long periods in the physical world without human intervention. From the governors controlling the speed of steam engines to complex guidance systems, these devices work as well as they do because they have the means for changing their actions in a way that depends on ***continuously*** sensed properties of their environments. I accept the central notion of control theory that continuous feedback is a necessary component of effective action.

Perhaps it is relatively easier for control theorists than it is for computer scientists to think about continuous feedback because control theorists conceive of their controlling

mechanisms as composed of electrical circuits or other physical systems rather than as automata with discrete read-compute-write cycles. The notions of goal-seeking servo-mechanisms, homeostasis, feedback, filtering, and stability-so essential to control in dynamic environments-arise rather naturally when one builds control devices with electrical circuits. Circuits, by their nature, are continously responsive to their inputs.

On the other hand, some of the central ideas of computer science, namely sequences, events, discrete actions, and subroutines, seem incompatible with the notion of continuous feedback. For example in conventional programming when one program calls another, the calling program is suspended until the called program returns control. This feature is awkward in applications in which the called program might encounter unexpected environmental circumstances with which it was not designed to cope. In such cases, the calling program can regain control only through interrupts explicitly provided by the programmer.

To be sure, there have been attempts to blend control theory and computer science. For example, the work of Ramadge and Wonham [Ramadge, 1989] on ***discrete-event systems*** has used the computer science notions of events, grammars, and discrete states to study the control of processes for which those ideas are appropriate. A recent book by Dean and Wellman [Dean, 1991] focusses on the overlap between control theory and artificial intelligence. But there has been little effort to import fundamental control-theory ideas into the core of computer science. That is precisely what I set out to do in this paper.

I propose a computational system that works quite differently than do those used previously in computer science. I also propose a language for writing control programs that has what I call ***circuit semantics;*** the execution of programs in this language produces (conceptually) electrical circuits, and it is these circuits that are used for control. [1] While importing control-theory concepts, I nevertheless want to retain useful ideas of computer science. The programs will have parameters that can be bound at run time and passed to subordinate routines, they can have a hierarchical organization, and they can be recursive. But since they are to be control programs, they must respond in bounded time to environmental changes.

Real-time control of agents in dynamic, uncertain environments is the subject of much recent attention. Several techniques and architectures have been proposed. I shall be comparing my approach to some of these and citing several relevant papers later. [2]

---

[1] Of course, the programs written by computer scientists run on computers made of circuits. But conventional programmers, while programming, use metaphorical constructs having quite different properties than do circuits.

[2] See, for example, the special issue on "Designing Autonomous Systems," of the journal *Robotics and Autonomous Systems,* 6, nos. 1 & 2, June, 1990 . Hanks and Firby [Hanks 1990] give a nice discussion of the problems.

In this paper, I shall not be discussing the larger issue of architectures for autonomous agents, although I think that robot control programs written in the language I am proposing will be important components of future architectures. Certain researchers, notably Laird and Rosenbloom [Laird, 1990], Maes, [Maes, 1989], Mitchell [Mitchell, 1990], and Sutton [Sutton, 1990], have proposed agent architectures that nicely integrate robot control with planning and/or learning programs. Here, I **assume robot control** programs are written by human programmers-not by planning systems. Even so, the form of the programs bears a striking resemblance to that of ***plans*** produced by various automatic planning systems, and toward the end of the paper I shall speculate on how one might exploit this resemblance to build and modify control programs by planning and learning methods.

Many of the ideas in this paper are elaborations of earlier work, some of which was done in collaboration with students [Nilsson, 1989; Nilsson, 1990a].

II. The Kaelbling-Rosenschein Model

Kaelbling and Rosenschein [Kaelbling, 1990] refer to "computer systems that sense and act on their environments" as ***embedded agents*** or ***situated automata.*** Their agents have a finite-state-machine structure much like that shown in Fig. 1. I base the semantics of my agent language on the circuitry used in this model, although I use rather different methods than do Kaelbling and Rosenschein to produce such circuitry.

Sensory input, and some of the agent's own actions, are processed by ***an update function*** that produces a ***state vector*** with binary-valued components. The update function performs, continously and in bounded time, the perceptual processing needed by the agent. In order to emphasize the point that the update function continuously delivers outputs from changing inputs, it is helpful to imagine that it is implemented by electrical circuitry. The outputs of this circuitry are available in the state vector. The designer of the agent will typically have in mind what each component in the state vector means (to the designer). In Fig. 1, for example, there is a component whose value is 1 only when the perceptual system, of a mobile robot say, determines that the path ahead is free of obstacles. (Of course, the perceptual system may be inaccurate.) The state vector may also contain information that depends upon previous actions and sensing. In Fig. 1, there is a component that is equal to 1 when the robot "believes" that the lights are on in room 17. So even though the robot might not have a sensor that gives it continously updated information about the status of the lights in room 17, it nevertheless can remember that they were recently determined to be on and are therefore likely to be on still. With our intended semantics, we (if not the agent) can think of the components of the state vector as the agent's ***beliefs*** about **its** world.

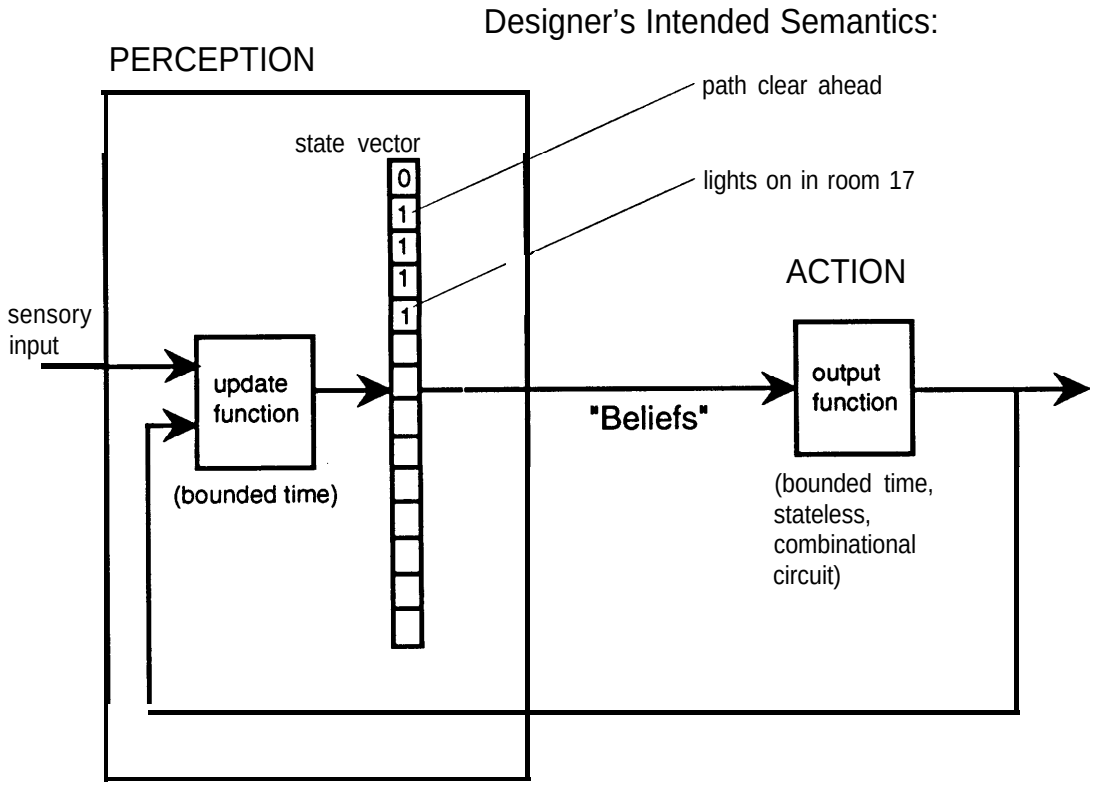


Figure 1. An Embedded Agent[3]

The components of the state vector, in turn, are inputs to ***an outputfunction,*** implemented by a combinational switching circuit, that energizes and sustains actions. These actions are typically ***durutive*** rather than discrete; that is, they continue so long as the output function maintains the values that sustain them. Some of the "actions" may change the values of certain components in the state vector; some have effects on the world; some perform sensory functions which ultimately affect the state vector. The output function can be implemented as a network of logical gates.

An example of a device adhering to these ideas is shown in Fig. 2; it moves a robot over a flat surface to a point denoted by the x-y coordinates given by the list loc. The update function ***continuously*** computes the predicates:

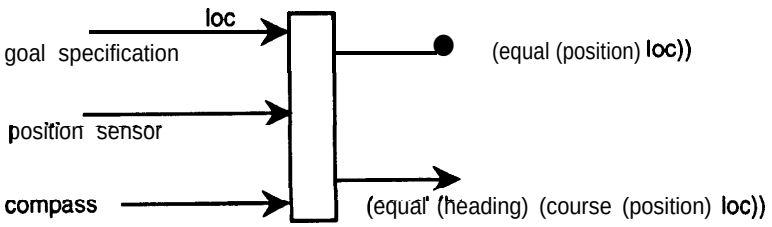
(equal (position) loc)

and

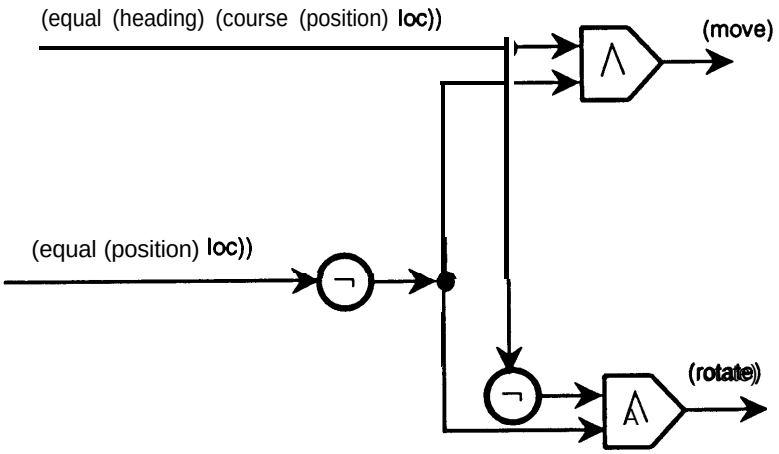
(equal (heading) (course (position) loc))

[3]Adapted from Kaelbling and Rosenschein [Kaelbling, 1990].

where (using LISP notation), position and heading are sensory functions that give the robot's current position and heading, respectively, and course is a function that computes the heading the robot ought to take to travel from its current position to loc. (In this simple example, the state vector does not store internal "state.") The action circuitry consists of AND and NOT gates that activate the primitive motor-action functions move and rotate whenever the robot is not at the goal location. (To avoid "hunting behavior" a practical circuit should include a "dead zone" in its predicates. Also, its update function should compute a direction-clockwise or counter-clockwise-that the robot should rotate depending on the robot's orientation with respect to the goal.)



Update Function



Output Function

Figure 2. Circuitry for Moving a Robot to the Point loc

Thebehavior of this device can best be described using a blend of the vocabularies of control theory and computer science. It exhibits goal-seeking and homeostatic behavior; it is stable only when the robot is at the goal point, loc. It unceasingly reacts to

continuous environmental signals. Yet, it has a hierarchical structure and uses parameters and functions (namely, loc, heading, position, and course) that are bound or computed at run time and continuously updated. The main program calls the primitive motor subroutines rotate and move. Note, however, that the called program move will give way to the program rotate whenever move's "precondition" (which is continuously being computed) is not satisfied.

Even though the steps of rotating and moving occur in sequence in this example, the "program" of Fig. 2 does not explicitly specify a sequence. The appropriate series of actions simply "emerges" from the interaction of the control system with its environment. Since action sequences are of prime importance in agent control, I base the language on a goal-seeking but yet reactive and emergent way of achieving them.

III. Teleo-Reactive Sequences and Trees

We desire to program agents to achieve various goals in their environments. These goals can be specified in terms of the components of the state vector. (The agent can only tell whether or not it has achieved its goal through its sensory apparatus, and therefore we have to specify goals in terms of the agent's sensory functions.) Usually more than one action is required to achieve a specified goal. The reasons for multiple actions are, first, that separate actions might be required to achieve different components of the goal, and, second, that some of the goal-achieving actions might not be executable in the agent's current environment-necessitating the prior performance of enabling actions. And so on. The agent determines that its actions have had their enabling or final effects by testing components of the state vector.

Sometimes an action does not have the effect that was anticipated by the agent's designer, and sometimes environmental dynamics (separate from the actions of the agent) change the world in unexpected (even helpful) ways. These phenomena, of course, are the reason continuous feedback is required. When the world is not completely predictable, the agent must constantly check to see which, if any, of its goals and preconditions for actions are satisfied.

We can imagine that such checking can proceed backwards from the goal condition. For example, the navigating robot of Fig. 2 should first check to see if it is already at the goal location. That is, it should check to see if (equal (position) loc)) is true. If so, it need (and should) cause no action. If not, it should check to see if it is perhaps heading in the appropriate direction:

(equal (heading) (course (position) loc))

If so, it should move (forward). If the heading is not correct, it should rotate. Although I have described the tests sequentially, they actually can be implemented, as they are in Fig. 2, by a non-sequential, combinational circuit.

These tests and actions can be represented as a path in a graph, as shown in Fig. 3. The nodes of the graph are labeled by conditions to be tested, and the arcs by actions that are to be energized. The bottom node is labeled by T representing true. The top node is labeled by the goal condition. I call such a path a ***teleo-reactive sequence*** because the program it represents is both goal-directed and ever-responsive to changing environmental conditions (as these conditions are sensed and then represented in the state vector).

We can imagine the graph of Fig. 3 as constituting a **program.**[4] Such a program is executed as follows: the interpreter looks for the shallowest true node and executes the action(s) on the arc exiting that node.

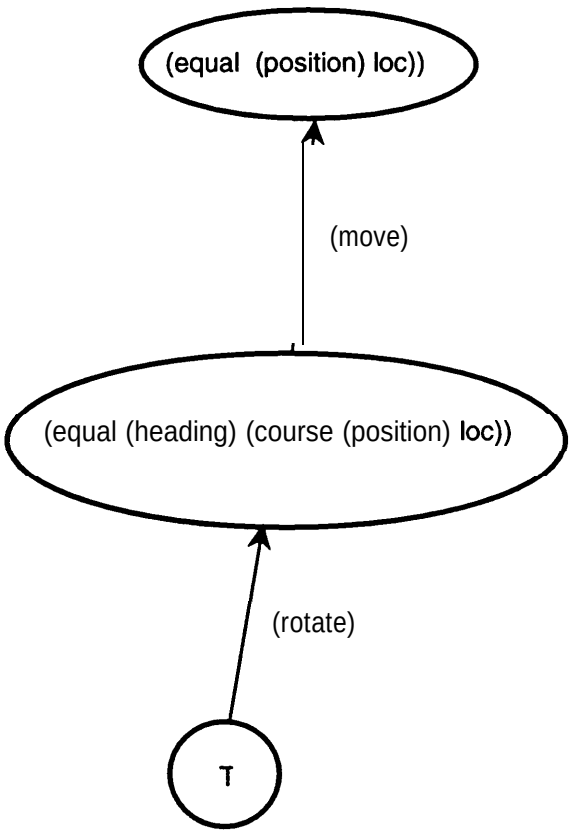
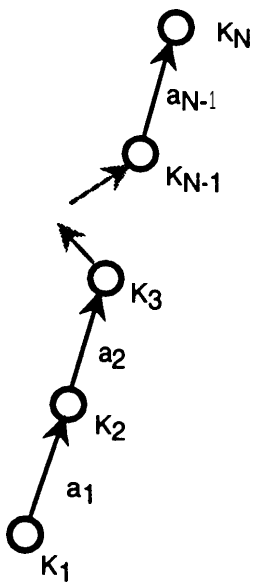


Figure 3. A Program for Controlling a Robot

[4] I am not explicitly advocating here a graphical programming language; the graphs in this section are pedagogic stepping stones to a language using conventional symbol strings that will be described later.

I show a general teleo-reactive (or T-R) sequence in Fig. 4. The programmer would ordinarily arrange: a) that each action in the sequence be executable if the condition at the tail of its arc is satisfied and b) that the execution of an action nominally achieve, eventually, the condition at the head of its arc. That is, as Fig. 4 indicates, the condition at a node is the ***regression*** of the condition at the node immediately above through the action labelling the arc connecting these nodes. Regression is defined in a way similar to that used in the automatic planning literature [Nilsson, 1980], namely it produces the weakest precondition that guarantees that the execution of the action will achieve the postcondition? In most of the automatic planning literature, however, actions are discrete and link discrete states. Here, I assume actions are continuous and will eventually produce their intended effect. Also, since I am not assuming (yet) that T-R sequences are produced by automatic planning systems, it is the human programmer who is (perhaps implicitly) computing the regressions as s/he writes the program.



The a_i are "actions"

The K_i are conditions

$K_{i-1} = \mathbf{Reg}(a_i, K_i)$

Figure 4. A Teleo-Reactive Sequence

[5] I use the word *guarantee* here loosely, of course, In dynamic, uncertain worlds there are no absolute guarantees; instead the *intended* effects of actions should be used in defining regression.

A T-R sequence is converted at run time to a combinational circuit, such as the circuit of AND and NOT gates shown in Fig. 5. It is for this reason that I say that my agent programs have ***circuit semantics;*** execution of the program produces a combinational circuit which is then connected to the agent's state vector and effectors to control the agent's actions. I shall discuss this point in more detail after I have given a more precise definition of T-R programs and how they are executed.6

In earlier work [Nilsson, 1990], I proposed the notion of ***an action unit.*** An action unit is a T-R sequence consisting of two nodes connected by a single action. T-R sequences are also related to the intermediate-level actions (ILA's) used in the SRI robot, Shakey, [Nilsson, 1984] and to triangle tables [Fikes, 1972]. (I will discuss the relationship with triangle tables in more detail later.) Neither the ILA's nor triangle tables had circuit semantics.

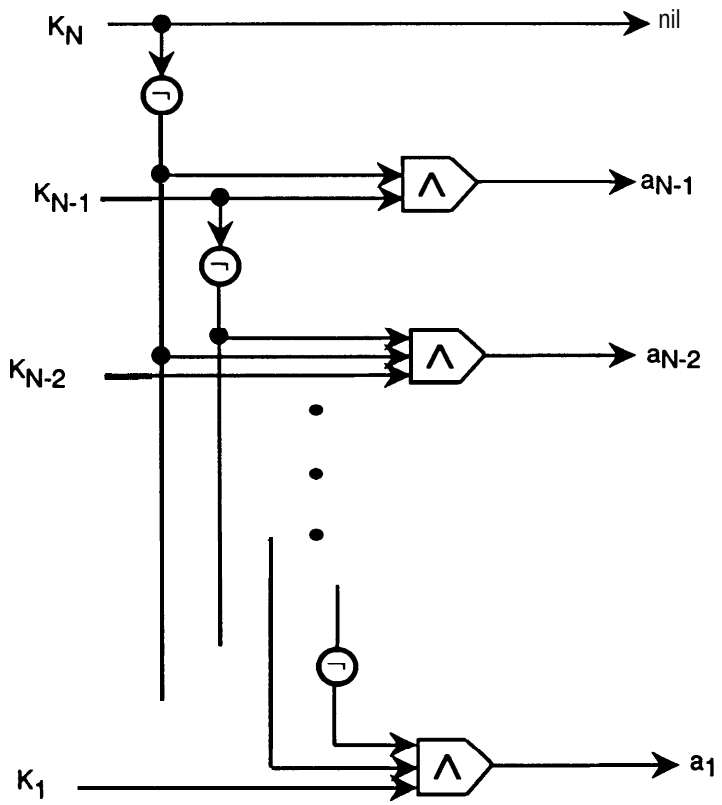


Figure 5. Implementing a T-R Sequence as a Circuit

6A question arises as to whether the circuit given in Fig. 5 is the "best" circuit for implementing the T-R sequence of Fig. 4. If, for example, there are repeated sub-expressions in the K_i, the circuit of Fig. 5 might have an equivalent with fewer gates. Since circuit optimization is a well understood subject, I will not deal with it here.

The reader might notice a similarity between T-R sequences and production systems. For example, it might appear possible to implement the sequence of Fig. 4 by the following list of ordered productions:

K_N —> nil
K_{N-1} —> a_{N-1}
...
K_2 —> a_2
K_1 —> a_1

Many production-rule interpreters work by scanning the productions in order, looking for the first true condition, and then executing the corresponding action. There are two major differences between T-R sequences and production systems. One is that most production systems are "flat" whereas the actions in a T-R sequence might themselves be coded as T-R sequences, etc. The more important difference is that T-R sequences have circuit semantics and production systems do not. For example, suppose action a_2 in Fig. 4 is activated (because K_2 is the shallowest true node). Perhaps before action a_2 finishes, K_3 becomes the shallowest true node. In that case, we want a_2 to be suspended and a_3 "called." In a production system, after a_2 was called, the conditions wouldn't even be checked again until a_2 finished and returned control. It is correct to think of T-R sequences as being hierarchical production systems with circuit semantics.

A teleo-reactive (T-R) tree is an obvious generalization of a T-R sequence. I illustrate one in Fig. 6. T-R trees are appropriate when there is more than one action that might contribute to achieving a given condition. Execution of a T-R tree proceeds in a fashion similar to the execution of a T-R sequence. We look for the shallowest true node—resolving depth ties according to some prespecified (possibly lexicographic) ordering.[7] With the ordering specified, a T-R tree can be converted to a T-R sequence (albeit one without the regression relationship linking adjacent elements of the sequence). For this reason, it is required only to provide a means for programming and implementing T-R sequences.

[7]Rather than looking for the *shallowest* true node, we might instead find that true node whose path from the goal is the least costly—assuming that all of the actions can be assigned costs. To simplify terminology, I shall nevertheless use the phrase "shallowest true node."

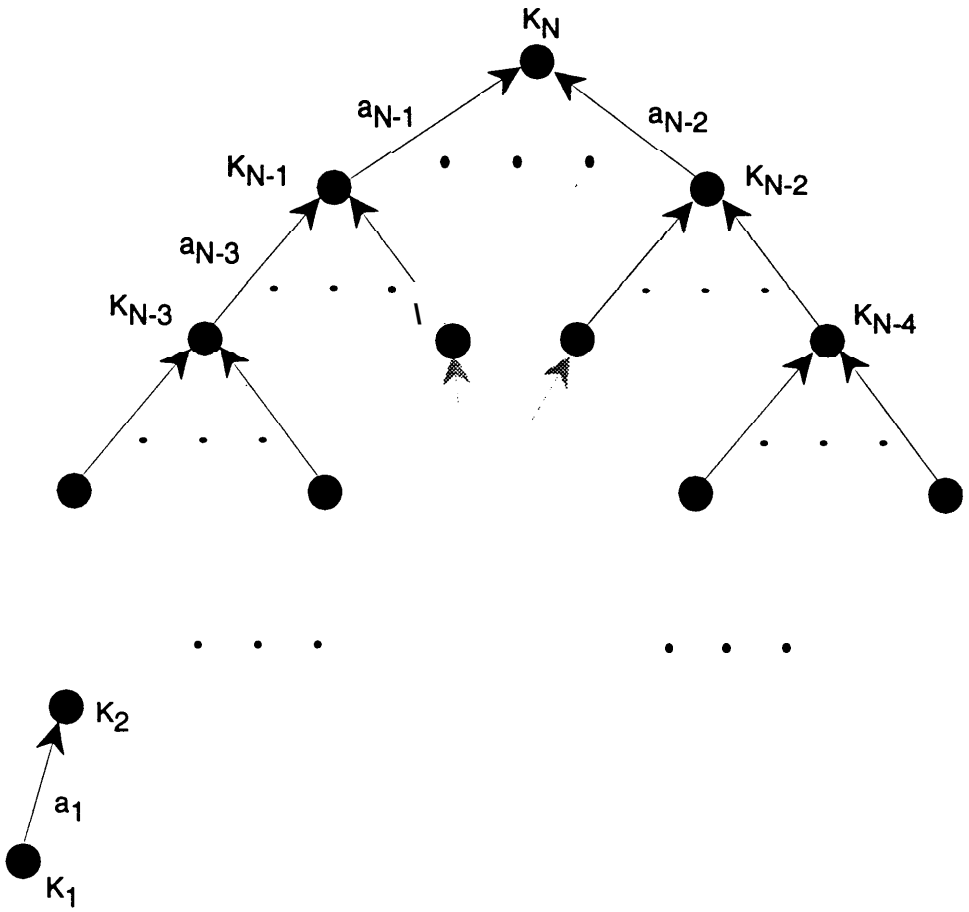


Figure 6. A T-R Tree

IV. Hierarchical Programs and Parallel Actions

The actions labeling arcs in a T-R sequence or tree can either be primitive, or they themselves might be coded as T-R programs. They might also be parallel combinations of actions. The condition at the tail of an arc labeling a list of parallel actions must be such that it is appropriate to execute the actions in parallel. That is, parallel actions must not interfere with each other.

I illustrate a hierarchical program containing parallel actions using an adaptation of Drummond's "B-not-last" (BNL) problem [Drummond, 1989]. In that problem, three

blocks, A, B, and C, must be placed, respectively at locations, 1, 2, and 3, as shown in Fig. 7. There are two primitive actions, which can under certain circumstances operate in parallel. The primitive action lplace can sweep a block into either positions 1 or 2 from the left if the target position and position 1 are "free". The action rplace can sweep a block into either positions 2 or 3 from the right if the target position and position 3 are free. Also, a block must be "available" in order for rplace or lplace to move it. Blocks are made available by some external process (perhaps, another agent). Block A becomes available to the left of position 1; block C becomes available to the right of position 3; and block B could come from either the left (if position 1 is free) or from the right (if position 3) is free. Note that block B cannot be placed last (after A and C). I assume perceptual functions for computing the needed predicates, free, at, and available.

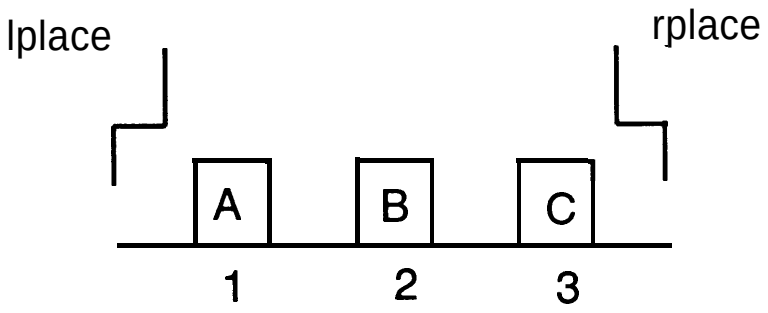


Figure 7. The BNL Problem

The programs, shown in Fig. 8, use the higher level parameterized programs rput and lput, which are themselves T-R programs that call the primitive rplace and lplace actions. I assume that the primitive programs cannot be interrupted while in progress; once called they run until they either succeed or fail. But rput and lput need to be sustained by some node in order to continue running. Note that there are circumstances under which rput and lput may be executed in parallel. Their execution results in the parallel execution of rplace and lplace if both of the blocks to be moved are available. Actions to be executed in parallel are indicated by lists. (As an abbreviation in Fig. 8, I have dropped some occurrences of the and connective; the condition at a node is assumed to be the conjunction of the individual predicates written in a node.)

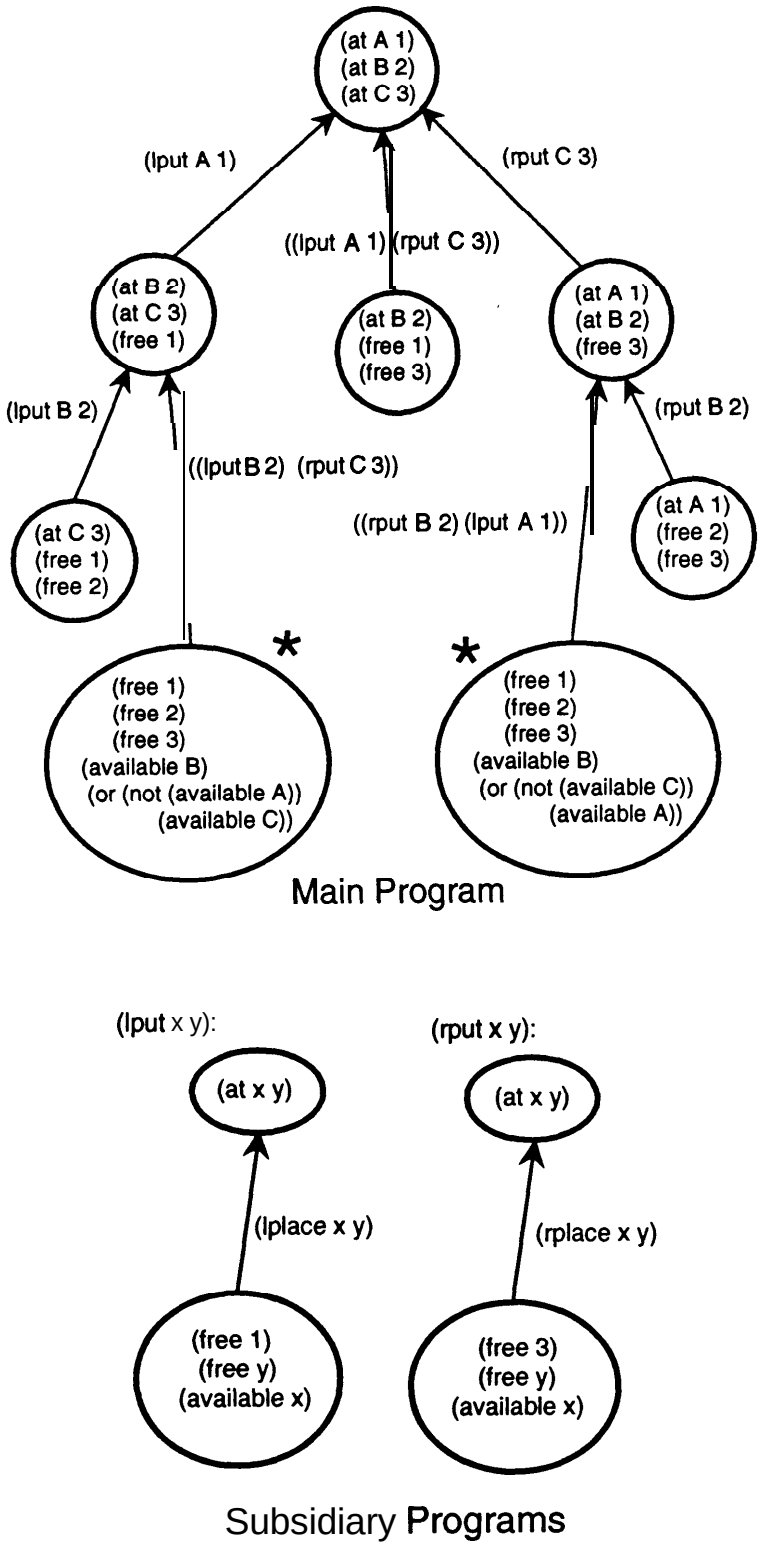


Figure 8. Programs for the BNL Problem

Interesting cases arise if all positions are free: the program will do nothing until some block becomes available. If, in that case, B is the only block available, both of the nodes marked with a * are true, and the program commits to one of these, say the left one, and executes the outgoing parallel actions. If A and *C* are available (whether or not B is) the program also commits to the left-most node marked with a *. If *C* is available, and B and A are not, then only the left-most node marked with a * is true. If A is available, and B and *C* are not, only the right-most is true.

The reader might want to imagine some of the various combinations of circumstances under which this program would operate and check to see that its actions are reasonable.

V. The T-R Language and its Interpreter

A. Syntax

Now that we have seen some examples of T-R programs I next propose a syntax for a programming language in which to write them. I give a partial definition of the language here and later point to a possible extension that might be useful.

The syntax for a T-R program is:

<*T-R-Prog*> :: < *T-R-Sequence*> I <*action-list*> I <*action*>

<*action-list*> :: (..., <*action*$_i$> . ..)

<*action*> :: <*energized-action*>|<*ballistic-action*>

<*energized-action*> :: <*primitive*>| <*T-R-Prog*>

<*ballistic-action*, : : < *primitive*

<*T-R-Sequence*> :: (defseq <*name*> <*arg-list*>

((<*K*$_N$> nil)

. . .

(<*K*$_i$> <*T-R-Prog*$_i$>)

. . .

(<*K*$_1$> <*T-R-Prog*$_1$>))

)

We see that a T-R program might consist of a single T-R sequence, an action, or of a list of actions. All of the actions in a list run concurrently and asynchronously in parallel. The individual actions may be ***energized*** or ***ballistic.*** Energized actions are the kind that must be sustained by an enabling condition to continue operating; ballistic ones, once called, run to completion. We-may as well assume that ballistic actions are primitive (for example, LISP code) since they need none of the T-R mechanisms. (Recall that the BNL problem used ballistic actions as primitives.)

A T-R sequence is defined using the construct defseq followed by a list of arguments that are bound when the program is called. A T-R program is "called" either as a top-level program or by virtue of some condition being true. Binding of arguments is, of course, looser than conventional binding; the value of the parameter to which an argument is bound is subject to continuous re-computation. The scope of the arguments includes the T-R program and all of its subsidiary programs (dynamic scoping). Then follows a list of condition-T-R program pairs. The conditions, K_i, are boolean combinations of components of a state vector as described previously. (Although this syntax looks something like a LISP cond statement, recall that the execution of a T-R sequence assumes circuit semantics. I will discuss how these programs are interpreted and executed in the next section.)

A T-R program for controlling a robot according to the scheme of Figs. 2 and 3 can be written thus:

```
(defseq goto (loc)
   (
     ((equal (position) loc) nil)
     ((equal (heading) (course (position) loc)) (move))
     (T  (rotate))
   )
)
```

The program takes an argument, loc, which although bound to an initial pair of x-y coordinates at run-time can be dynamically changed during running. The values of the subsidiary functions position and heading are continuously being updated by sensors—even while goto is running, and goto always uses the ***current*** values! The function course continuously computes the direction the robot should take to go from the current value of position to the current value of loc. The functions move and rotate energize the appropriate robot motors.

This example illustrates that it would be useful to allow the use of local variables within T-R sequences. One way to do this would be to use a construction similar to the Common LISP let*. With that modification, our program would become:

```
(defseq goto (loc)
  (let* (
         (pos  (position))
         )
    (
      ((equal pos loc) nil)
      ((equal (heading) (course pos loc)) (move))
      (T  (rotate))
    )
  )
)
```

Unlike the case in LISP, however, the values of the local variables must be continuously computed.

B. Program Interpretation and Circuit Simulation

Even though I have already informally discussed how T-R programs are to be executed, there are some additional points to be made. Execution of a T-R program involves the dynamic interaction of an interpreter and a circuit simulator. I give here a brief summary of essentially what the interpreter and simulator must accomplish.

When a program is called (by the user or by another program), the interpreter binds the parameters in the program's definition and creates circuitry similar to that shown in Fig. 5. The values of any of the program's local variables and predicates (the K_i) are computed from the basic predicates and terms produced by the perceptual processes. The circuit thus produced is then simulated by the simulator. The action called for by the circuit (if any) is then energized. If this action is itself a T-R program, the interpreter binds its parameters, additional circuitry is produced, and the process continues until some primitive action is energized. Everything that happens up until the first primitive action is energized is called a ***simulation cycle.***

The interpreter/simulator then waits for some pie-specified, small ***sampling interval*** during which time whatever primitive actions are being executed have had some incremental effect on the world, and the sensing apparatus has updated the state vector. At this time, another simulation cycle begins: the values of all predicates and functions in the networks are re-computed as necessary, actions are computed by the circuitry, and additional network is created until another primitive action is energized. Thus, at all levels, the circuitry operates at the beginning of each sampling interval-simulating continuous operation. Note that primitive actions active during one sampling interval might not be active during the next. At any level, sustained activation depends on the appropriate predicates being true. Typically, I assume that the time required for a simulation cycle is short compared to a sampling interval which is short compared to the pace of events in the world.

An efficient implementation of the interpreter/simulator need only recompute at the end of each sampling interval those functions whose values depend on sensed or stored data that have changed since the last sampling interval. Assumption-based truth maintenance systems [deKleer, 1986] would be useful in this regard.

In the example robot control program, goto, all of the actions specified in the program are primitive, so the interpreter constructs a circuit similar to that shown in Fig. 2 by the end of the first simulation cycle, and no more circuitry need be created.

When a T-R program calls a non-primitive program, the situation is a bit more complicated. Consider the following recursive program:

```
(defseq amble (loc)
   (
     ((equal (position) loc) nil)
     ((clear (position) loc) (got0 loc))
      (T (amble (new (position) loc)))
   )
)
```

The subsidiary sensory function clear checks to see if there is a clear path between the robot's current position and its goal location. If there is a clear path, the function goto is called. If not, amble is called recursively with a new location computed using the sensory function new, which we can assume selects some sub-goal location on the way around the obstacle perceived to lie between the robot's current position and the goal.

The circuit constructed by the interpreter depends on the configuration of robot, obstacle (if any), and goal. In Fig. 9, I show an early stage of circuit construction.

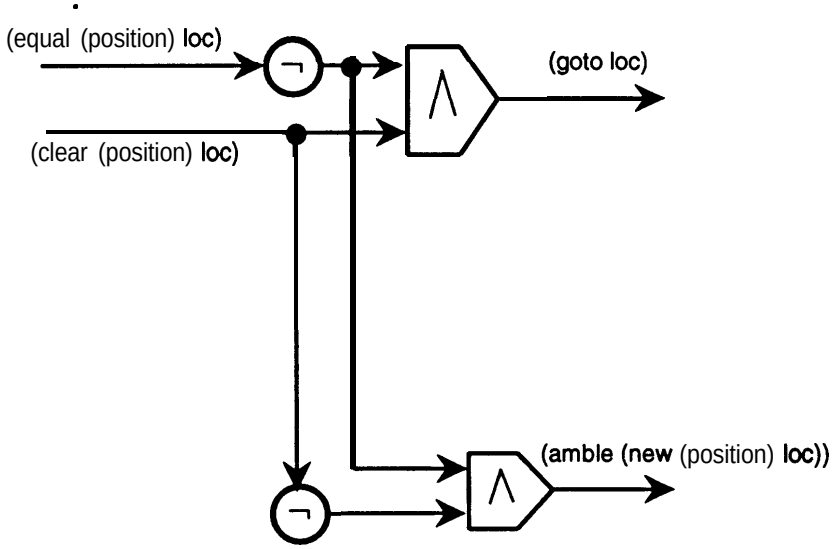


Figure 9. Early Stage of Circuit Construction

If the robot is not already at loc, and if there is a clear path to loc, then (goto loc) is interpreted, and more circuitry is built as in Fig. 10.

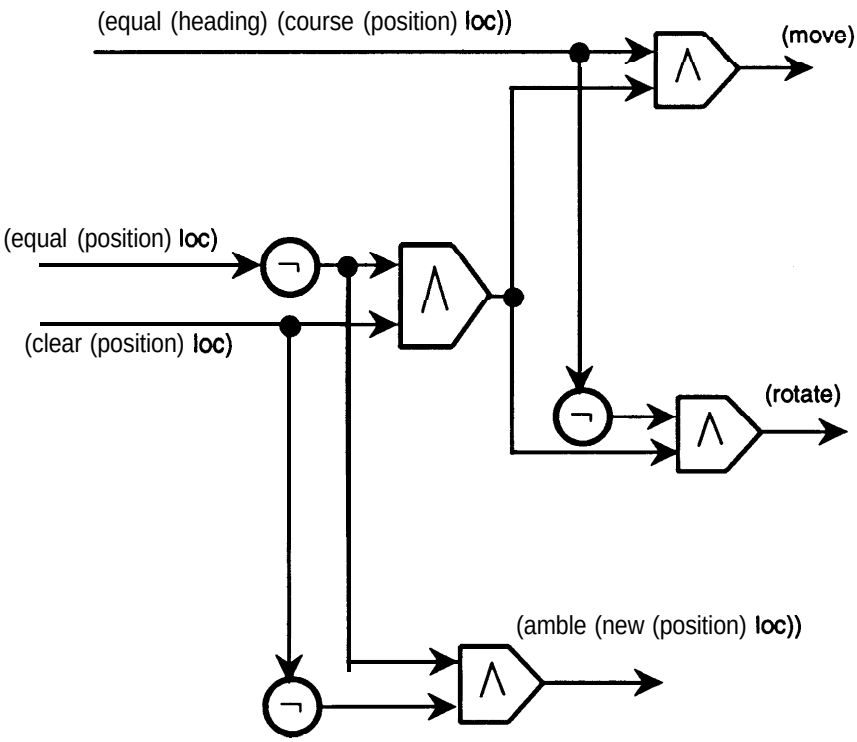


Figure 10. The Circuit at the End of a Simulation Cycle

Now, if the value of heading is not equal to the value of course, the primitive action (rotate) is selected by this circuitry--ending a simulation cycle. The circuit of Fig. 10 continues to control the robot so long as there is a clear path to loc.

If the path to loc becomes blocked for some reason, the circuitry of Fig. 10 calls for the execution of the non-primitive program (amble (new (position loc)), which requires interpretation and another simulation cycle-resulting in the circuit of Fig. 11. Under the

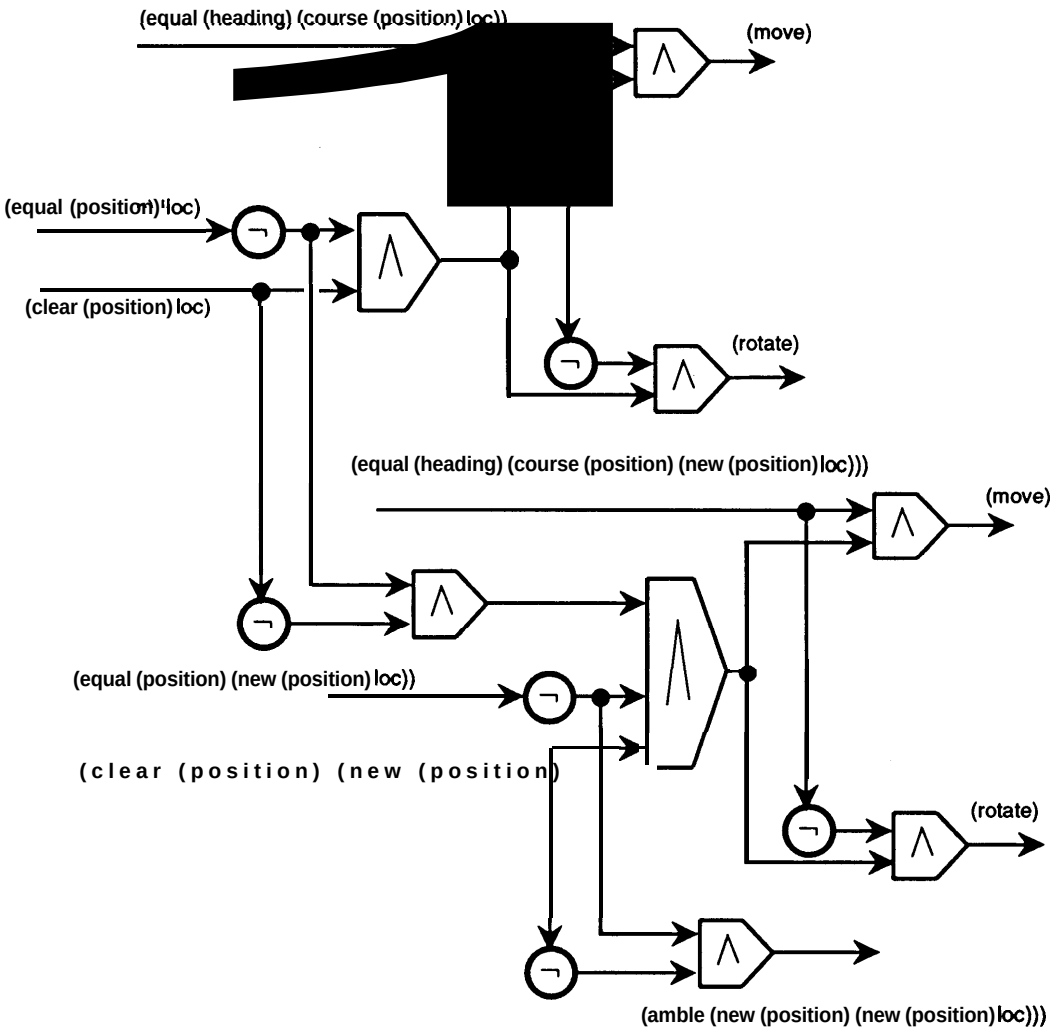


Figure 11. The Circuitry for Navigating Around an Obstacle

control of this circuit, the robot will detour around a single obstacle until the path to loc is clear. If another obstacle blocks the detour, then another recursive call to amble is made.

(The reader should understand that I am not necessarily recommending amble as an actual robot navigation strategy; I use it simply as an example to illustrate how recursive T-R programs are interpreted.)

VI. Preliminary Experiments

I have carried out some early experiments with agents programmed in this language. Although an interpreter/simulator for T-R programs that works precisely as described in the previous section has not yet been implemented, we can use LISP cond functions (with primitive action increments and short sampling intervals) to simulate the execution of T-R programs. (In earlier work [Nilsson, 1990a] an interpreter/simulator for a related language with circuit semantics was constructed.)

The agents I am working with are simulated robots acting in a two-dimensional space, called Botworld[8], of construction materials, structures made from these materials, and other robots. I show a scene from this world in Fig. 12. The construction materials are

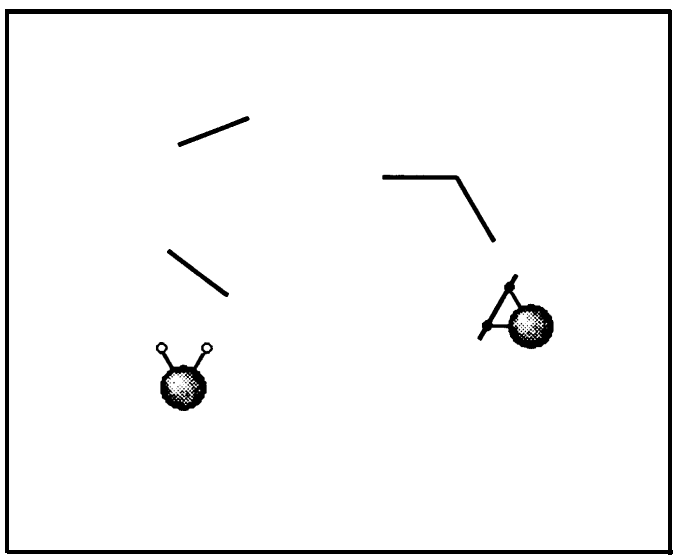


Figure 12. Botworld

bars, and the robots are to build structures by connecting the bars in various ways. A robot can turn and move, can grab and release a suitably adjacent bar, can turn and move

[8]The original Botworld interface, including the primitive perceptual functions and actions for its robots, was designed and implemented by Jonas Karlsson for the NeXT computer system [Karlsson, 1990]. Subsequently, Patrick Teo has implemented a version that runs under X-windows on any of several different workstations [Teo, 1991]. The latter version allows the simulation of several robots simultaneously—each under the control of its own independently running process.

a grabbed bar, and can connect a bar to other bars or structures. The robots continuously sense whether or not they are holding a bar, and they "see" in front of them (giving them information about the location of bars and structures). Because of the existence of other robots which may change the world in sometimes unexpected ways, it is important for each robot to sense certain critical aspects of its environment continuously.Some of the T-R programs (in graphical form) for controlling a robot in grabbing a bar are shown in Fig. 13. The primitive actions, in LISP notation, are (grab bar), (move), (turn-cw), and

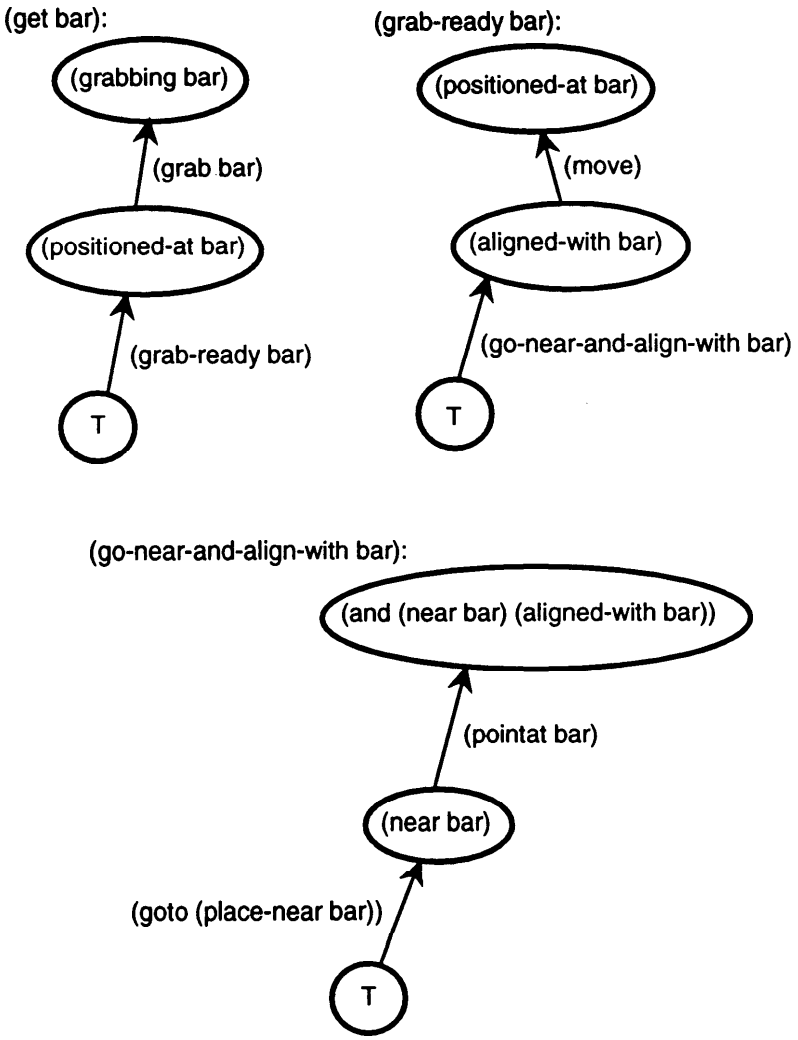


Figure 13
. Some T-R Programs for Grabbing a Bar

(turn-ccw). The programs pointat and goto are themselves T-R programs (similar to but somewhat more complex than the one illustrated in Fig. 3). The predicates and terms used are defined roughly as follows:

grabbing the bar named bar. (Bars are named by their position-orientation coordinates.)

(positioned-at bar): a predicate which is true if the robot is within the grabbing distance of bar and facing it

(aligned-with bar): a predicate which is true if the robot is facing bar and positioned on a line perpendicular to it

(near bar): a predicate which is true if the robot is positioned on a line perpendicular to bar and within a certain distance of it

(place-near bar): a computed location from which the robot can move toward bar to grab it

A robot controlled by this program exhibits interesting homeostatic behavior. So long as the robot is grabbing bar the system is stable. If a grabbed bar is taken from a robot under the control of this program, the robot will become active and persist until it grabs it again. If the target bar moves while the robot is heading toward it, the robot will appropriately correct its own course (although I have not yet implemented programs that try to predict the bar's course).

The programmer need only be concerned with inventing the appropriate predicates from the available perceptual functions and how they are to be used as goals and subgoals. S/he does not need to worry about providing interrupts of lower level programs so higher level ones can regain control. Debugging T-R programs presents some challenges, though. Since they are designed to be quite robust in the face of environmental uncertainty, they also sometimes work rather well even though they are not completely debugged. These residual errors might not have undesirable effects until the programs are used in higher level programs—making the higher ones more difficult to debug.

VII. Other Approaches for Specifying Behavior

There have been several formalisms proposed for prescribing sensory-directed, real-time activity in dynamic environments. Many of these are closely related to the T-R language proposed here. In this section I point out the major similarities and differences between T-R programs and a representative, though not complete, sample of their closest relatives.

A. Production Systems

As has already been mentioned, production systems [Waterman, 1978] with ordered production rules look very much like T-R programs. The intermediate-level actions (ILAs) used in the robot Shakey [Nilsson, 1984] were programmed using production rules and were very much like T-R sequences except that the ILAs did not have circuit semantics. Neither do other production systems. Also a programmer would ordinarily

write a T-R program in such a way that the actions specified by the i-th rule would typically cause the condition in a rule higher in the ordering to become true.

B. Situated Automata and Other Circuit-Based Systems

Kaelbling has proposed a formalism called **GAPPS** [Kaelbling, 1988; Kaelbling, 1990], involving ***goal reduction*** rules, for implicitly describing how to achieve goals. The **GAPPS** programmer defines the activity of an agent by providing sufficient goal reduction rules to connect the agent's goals with the situations in which it might find itself. These rules are then compiled into circuitry for real-time control of the agent. Rosenschein and Kaelbling [Rosenschein, 1986] call such circuitry ***situated automata.***

A collection of GAPPS rules for achieving a goal can be thought of as an implicit specification of a T-R tree in which the computations needed to construct the tree are performed when the rules are compiled. The **GAPPS** programmer typically exerts less specific control over the agent's activity—leaving some of the work to the search process performed by the GAPPS compiler. For example, a T-R tree to achieve a goal, p, can be implicitly specified by the following **GAPPS** rule:

```
(defgoalr (ach ?p)
        (if ((holds ?p) (do nil))
            ((holds (regress ?a ?p)) (do ?a))
            (T ach (regress ?a ?p)) ))
```

The recursion defined by this rule bottoms out in rules of the form:

```
(defgoalr (ach φ)
        ((holds ψ) (do α)))
```

where ϕ and ψ are conditions and ***a*** is a specific action.

The compiling process needed to transform **GAPPS** rules into circuitry might exceed the time bounds required for real-time control in dynamic worlds, and for this reason compilation must be done before run time. Pre-run-time compiling means that more circuitry must be built than might be needed in any given run because all possible contingencies must be anticipated at compile time. Since T-R programs already provide an explicit (though highly conditional) specification for action, they can be interpreted to create just the circuitry needed in bounded time at run time. The two formalisms do share, however, the important notion of using circuitry for run-time control.

In implementing a system to play a video game, Chapman [Chapman, 1990] compiles production-like rules into digital circuitry for real-time control using an approach that he calls "arbitration macrology." As in situated automata, the compilation process occurs prior to run time.

C. Situated Control Rules

Drummond [Drummond, 1989] introduces the notion of a *plan net* which is a kind of Petri net [Reisig, 1985] for representing the effects of actions (which can be executed in parallel). Taking into account the possible interactions of actions, he then projects the effects of all possible actions from a present state up to some horizon. These effects are represented in a structure called a ***plan projection.*** The plan projection is analyzed to see, for each state in it, which states ***possibly*** have a'path to the goal state. This analysis is a forward version of the backward analysis used by a programmer in producing a T-R ***tree. Situated control rules*** are the result of this analysis; they constrain the actions that might be taken at any state to those which will result in a state that still possibly has a path to the goal. But they don't guarantee a shortest (or least costly) path, and neither they nor Petri nets embody circuit semantics.

D. Reactive Plans

Several researchers have adopted the approach of using the current situation to index into a set of prearranged action sequences. This set can either be large enough to cover a substantial number of the situations in which an agent is likely to find itself or it can cover all possible situations. In the latter case, the plan set is said to ***be universal*** [Schoppers, 1987]. Georgeff and Lansky [Georgeff 1987] have implemented a system called **PRS** in which plans for action sequences can be cached and those appropriate to the current situation can be retrieved and executed. Firby [Firby, 1987] has proposed modules called "reactive action packages." The "routines" and "running arguments" of Agre [Agre, 1989] are of a similar character. As with T-R sequences, time-space trade-offs must be taken into account when considering how many different conditions must be anticipated in providing reactive plans. Ginsberg has noted that in several domains, the number of situations likely to be encountered by the agent is so intractably large that the agent is forced to postpone most of its planning until run time when situations are actually encountered [Ginsberg, 1989].

E. Other Languages and Formalisms

Brooks has developed a behavior language, **BL** [Brooks, 1989], for writing reactive robot control programs based on his "subsumption architecture" [Brooks, 1986]. A similar behavior description language, **BDL,** has been implemented by Gat and Miller [Gat, 1990]. Programs written in these languages compile into structures very much like circuits whose action computations are performed in bounded time. Again, compilation occurs prior to run time. It has been relatively straightforward to translate examples of subsumption-architecture programs into the T-R tree formalism, although I think my run-time interpretation has advantages over their design-time compilation.

ESTEREL [Berry, 1983] is a computer language whose programs compile into finite-state machines. These machines are inserted into the environment along with the processes they control. The **ESTEREL** formalism is based on discrete states and events in contrast to the durative actions and circuit semantics of the T-R language. ACORE manning, 1989] is a language for writing asynchronous and concurrent programs that have many features in common with T-R programs.

Research has also been directed at problems in which. the dynamic environment itself can be either fully or partially described as an automaton; control actions can then be based on anticipated environmental conditions. For a representative, though incomplete, sample see: [Pneuli, 1989; Ramadge, 1989; Lyons, 1990; and Harel, 1988].

VIII. Automatic Planning and Learning

In this section I suggest some ideas for incorporating planning and learning systems in agents using T-R programs. Although these ideas are relatively straightforward, their implementation in a complete agent architecture would require extensive additional work.

A. Planning

A T-R sequence is much like a plan represented in triangle-table form constructed by **STRIPS** [Fikes, 1972]. Each of the K_i of a T-R sequence corresponds to a triangle table kernel. In the **PLANEX** execution system for triangle tables, the action corresponding to the highest-numbered satisfied kernel is executed. The major difference between T-R sequences and triangle tables is that T-R sequences have circuit semantics and triangle tables do not. This similarity suggests that T-R sequences could be automatically constructed by an automatic planning system suitably modified to deal with durative (rather than discrete) actions. This modification may require that the planning system use a temporal logic in order to be able to represent the effects of durative actions.

Both triangle tables and T-R sequences can be thought of as representing solution paths in a search space generated by a backward-directed planning system. After the planner completes the search and finds a solution path, this path can be converted into either a triangle table or a T-R sequence. A T-R tree can similarly be thought of as the entire search tree (including the solution path) produced by a regression-based planner. Thus, a T-R tree for achieving some particular goal can be generated by a backward-directed automatic planning system. (The idea of a triangle table can be generalized to a triangle-table tree [Nilsson, 1989]. I have also proposed a scanning algorithm for triangle table trees [Nilsson, 1990b].) I believe that one of the potential advantages of the T-R formalism (over other reactive languages) is this similarity between T-R trees and planning search trees.

B. Learning

I imagine that a human programmer might initially supply some T-R programs for an

agent and that automatic planning and learning systems would incrementally modify and add to these programs. There are two major issues to discuss related to such incremental modification. One is coverage, and the other is accuracy. The coverage problem concerns the fact that the human programmer might not anticipate the precise set of situations or goals that the agent will ultimately face. I would like the automatic planning system to be able to add and to modify programs as new goals and situations are encountered. The accuracy problem concerns the fact that the human programmer might not have a sufficiently correct model of the agent's actions-neither as s/he has used them in hand-coded T-R programs nor as described to the planner. One might attempt to apply machine learning techniques to the problem of learning more accurate descriptions while the agent is actually behaving in its world. These modified descriptions would then be used by the planner in subsequent T-R program generation.

I discuss the matter of T-R program generation and modification first. Suppose there is no T-R program available in the agent's repertoire of cached programs to achieve a goal given to the agent. In this case the planning system can be called to generate one, and a T-R program can be produced from the resulting search tree. Or suppose, there is a cached T-R program but no conditions in that program are true. That is, there are no true nodes in the corresponding T-R tree. In this case it is as if the search process employed by an automatic planner had already produced this T-R tree but had not yet terminated because no subgoal was already satisfied in the current state. In this case the planning system can be called upon to continue to search; that is, the T-R tree will be expanded until a true node is produced.

In both of these processes, explanation-based generalization [Mitchell, 1986] (similar to the version of it used for substituting variables for constants in creating generalized triangle tables [Fikes, 1972]) can be used to generalize the new trees and tree portions.

The reader might object that there is no reason to suppose that the search trees produced by an automatic planning process will contain nodes whose conditions are those that the agent is likely to encounter in its behavior. The process for incremental modification of the T-R trees, however, should gradually make them more and more matched to the agent's environment. Trees are modified by the planning system in response to situations not represented in the existing trees. The trees can be made to match the agent's needs even more by keeping statistics on how often their nodes are satisfied. Portions of the trees that are never or seldom used can be erased. Such a process would, of course, have to be sensitive to the relative costs of storage space versus computation time.

Suppose next that the action descriptions implicitly represented in the T-R trees and explicitly given to the automatic planner might be too often incorrect. In such a case, the node at the head of an action arc will not usually be the next shallowest true node after executing that action. Such a possibility prompts us to keep statistics on the efficacy of actions in a T-R tree. Suppose we associate with each arc a_i in the tree a number q_i that is equal to the fraction of the cases in which the activation of action a_i on that arc led to

the activation of the action associated with the expected next arc in the tree. Then, if q_i fell below some threshold, the sub-tree hanging below arc a_i could be pruned from its current position and perhaps reattached in a more appropriate place.

In addition to keeping statistics on how often the action associated with arc a_i led to the expected next action, we can create ***shadow arcs*** between nodes in the tree that are sequentially the shallowest true nodes and keep statistics on these shadow arcs. When a shadow arc becomes sufficiently strong (which will happen at the same time or after the actual arc becomes sufficiently weak), we can reattach the corresponding tree portion using the shadow arc.

I give examples of these changes to a T-R tree in Fig. 14.

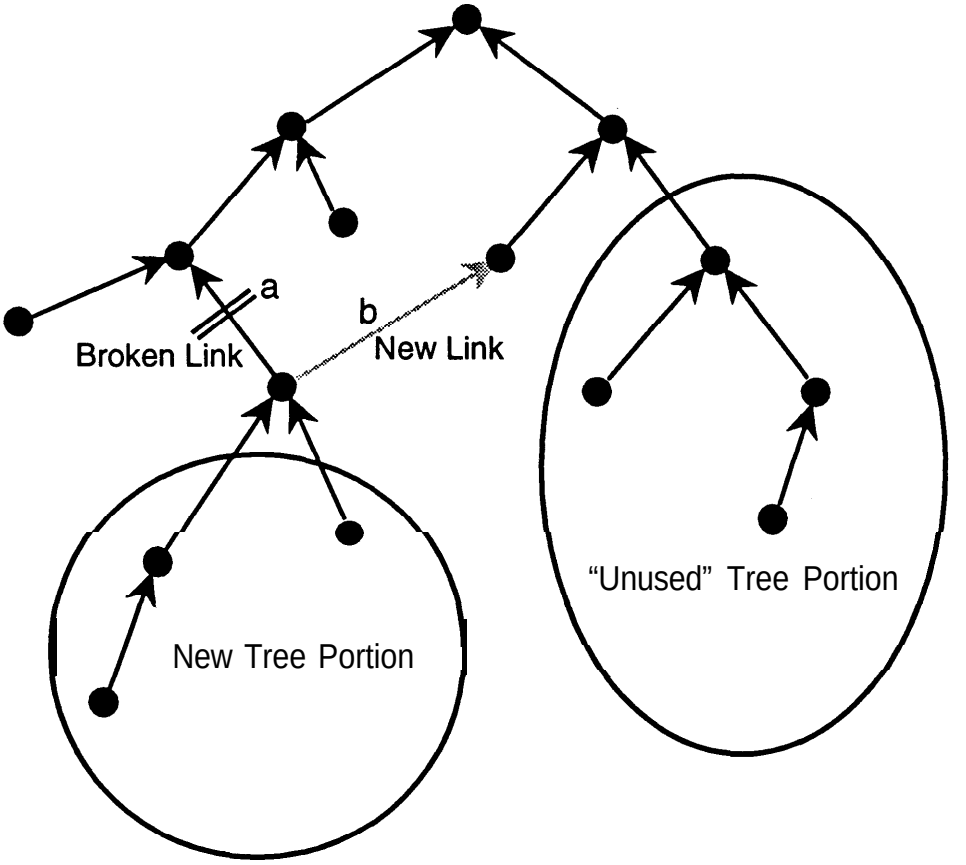


Figure 14. Modifications to a T-R Tree

In Fig. 14, I show a portion of the T-R tree that can be removed because it has been relatively unused and another portion that is added by a planning system. After adding the new subtree we might break link a and add link b.

Of course, when a tree portion is re-attached, we have evidence that the corresponding STRIPS rule is inaccurate. We must also invent mechanisms for changing this rule. Although I will not deal with this matter in this paper, perhaps techniques similar to those suggested by Vere [Vere, 1978] for learning relational production systems and/or Shen [Shen, 1989] for learning the effects of actions will be useful in this regard. (See also [Christiansen, 1991].)

IX. Conclusions and Future Work

I have presented a formalism for specifying actions in dynamic and uncertain domains. Since this work rests on ideas rather different than those of conventional computer science, I expect that considerable more analysis and experimentation will be required before the T-R formalism can be fully evaluated. The need in robotics for control-theoretic ideas such as homeostatis, continuous feedback, and stability appears to be sufficiently strong, however, that it seems appropriate for candidate formalisms embodying these ideas to be put forward for consideration. An appropriate next step would be the specification of a comprehensive language based on the T-R formalism and the implementation of an interpreter and circuit simulator for this language.

The language ought to embody some features that have not been discussed in this paper. Explicit reference to time in specifying actions might be necessary. For example, we might want to make sure that some action a is not initiated until after some time t_1 and ceases after some time t_2. Time predicates, whose time terms are evaluated using an internal clock, may suffice for this purpose. Also, in some applications we may want to control which nodes in a T-R tree are actually tested. It may be, for example, that some conditions won't have to be checked because their truth or falsity can be guessed with compelling accuracy.

Experiments with the language will produce a stock of advice about how to write T-R programs effectively. Already, for example, it is apparent that a sustaining condition in a T-R sequence must be carefully specified so that it is no more restrictive than it really needs to be; an overly restrictive condition is likely to be rendered false by the very action that it is supposed to sustain before that action succeeds in making the next condition in the sequence true.

It will be interesting to ask in what sense T-R programs can be proved to be "correct." It would seem that any verification analysis would have to make assumptions about the dynamics of the environment; some environments might be so malevolent that agents in them can never achieve their goals.

The question of what constitutes a *goal* is itself a matter that needs further development. I have assumed goals of achievement. Can mechanisms be found that continously avoid making certain conditions true (or false) while attempting to achieve others? Or suppose priorities on a number of possibly mutually contradictory conditions are specified; what are reasonable methods for attending to those achievable goals having the highest priorities?

I have already speculated about how to integrate planning and learning methods in an agent architecture that uses T-R programs. Besides extending planning programs to handle durative actions, we will want them to be able to plan parallel actions. For a backward-directed planner this ability would seem to require regressing conditions through parallel actions.

Lastly, I think that it would be fruitful to inquire whether temporal-difference, delayed-reinforcement learning algorithms [Sutton, 1990] could be used to generate or modify T-R programs. Related work on learning robot control programs using delayed-reinforcement methods appears promising [Mahadevan, 1990; Lin, 1991].

X. Acknowledgements

I trace my interest in reactive, yet purposive, systems to my early collaborative work on triangle tables and ILAs. The present manifestation of these ideas owes much to several Stanford students including Jonas Karlsson, Eric Ly, Rebecca Moore, and Mark Torrance. I also want to thank Matt Ginsberg, Leslie Kaelbling, and Karen Myers for useful comments about earlier versions of this paper. I gratefully acknowledge the supportive settings provided during part of a sabbatical year by Harvard University and Prof. Barbara Grosz; and the Massachusetts Institute of Technology and Prof. Rodney Brooks. Pattie Maes and Maja Mataric at MIT gave me many helpful suggestions. A week spent at Carnegie-Mellon University with Prof. Tom Mitchell and colleagues was also very enlightening.

This work was performed under NASA Grant NCC2-494; Monte Zweben, Mark Drummond, and Peter Friedland at NASA-Ames supplied valued constructive criticism.

REFERENCES

Agre 1989
Agre, P., ***The Dynamic Structure of Everyday Life***, MIT AI Lab Technical Report TR 1085, 1989. (Also to be published by Cambridge University Press.)

Berry 1983
Berry, G., Moisan, S., and Rigault, J. P., "ESTEREL: Towards a Synchronous and

Semantically Sound' High Level Langauge for Real-Time Applications," *Proc. IEEE 1983 Real-Time Systems Symposium, 1983.*

Brooks 1986
Brooks, Rodney A., "A Robust Layered Control System for a Mobile Robot," ***IEEE Journal of Robotics and Automation,*** March 1986.

Brooks 1989
Brooks, R. A., "The Behavior Language User's Guide," MIT Artificial Intelligence Laboratory Seymour Implementation Note **#2,** October, 1989.

Chapman 1990
Chapman, D., ***Vision, Instruction and Action,*** MIT AI Lab Technical Report 1204, Massachusetts Institute of Technology, April, 1990 .

Christiansen 199 1
Christiansen, A., Mason, M., and Mitchell, T., "Learning Reliable Manipulation Strategies without Initial Physical Models," ***Robotics and Autonomous Systems, 1991.***

deKleer 1986
de Kleer, J., "Problem Solving with the ATMS," ***Artificial Intelligence, 28,*** pp. 197-224, March, 1986.

Dean 1991
Dean, T., and Wellman, M., ***Planning and Control,*** Morgan Kaufmann, San Mateo, CA 1991

Drummond1989
Drummond, M., "Situated Control Rules," ***Proc. First International Conf. on Principles of Knowledge Representation and Reasoning,*** Toronto, May 1989, Morgan Kaufmann, San Mateo, CA 1989.

Fikes 1972
Fikes, R., Hart, P., and Nilsson, N., "Learning and Execution of Generalized Robot Plans," ***Artificial Intelligence, 3, 1972.***

Firby 1987
Firby, R., "An Investigation into Reactive Planning in Complex Domains," *Proc. AAAI-87,* Morgan Kaufmann Publishers, San Mateo, CA, 1987.

Gat 1990
Gat, E., and Miller, D. P., "BDL: A Language for Programming Reactive Robotic Control Systems," unpublished paper, California Institute of Technology/Jet Propulsion Laboratory, 1 June 1990.

Ginsberg 1989
Ginsberg, M. L., "Universal Planning: A (n Almost) Universally Bad Idea," ***AAAI Magazine,*** 10, no. 4, pp 40-44, Winter, 1989.

Georgeff 1987
Georgeff, M., and Lansky, A., "Reactive Reasoning and Planning," *Proc.* AAAI-87, Morgan Kaufmann Publishers, San Mateo, CA, 1987.

Hanks 1990
Hanks, S., and Firby, R. J., "Issues and Architectures for Planning and Execution," in Sycara, ***K.*** (ed.), *Proc.* ***Workshop on Innovative Approaches to Planning, Scheduling, and Control,*** Defense Advanced Research Projects Agency, November, 1990, pp. 59-70, Morgan Kaufmann, San Mateo, CA, 1990.

Hare1 1988
Harel, D., "On Visual Formalisms," *CACM,* **31,** no. 5, pp. 514-530, May, 1988.

Kaelbling 1988
Kaelbling, L. P., "Goals as Parallel Program Specifications," Proceedings AAAI-88, Saint Paul, MN, pp. 60-65, American Association for Artificial Intelligence, 1988.

Kaelbling 1990
Kaelbling, L. P., and Rosenschein, S. J., "Action and Planning in Embedded Agents," ***Robotics and Autonomous Systems,*** 6, nos. 1 & 2, pp. 35-48, June, 1990.

Karlsson 1990
Karlsson, J., "Building a Triangle Using Action Nets," unpublished project paper, Stanford Computer Science Dept., June 12, 1990.

Laird 1990
Laird, J. E., and Rosenbloom, P. S., "Integrating Execution, Planning and Learning in Soar for External Environments," *Proc. AAAI-90,* pp. 1022-1029, July, 1990.

Lin 1991
Lin, Long-Ji, "Programming Robots Using Reinforcement Learning and Teaching," *Proc.* ***AAAI-91,*** AAAI Press, I99 1.

Lyons 1990
Lyons, D., "A Formal Model for Reactive Robot Plans," Philips TR-90-038, Autonomous Systems Dept., Philips Laboratories, 345 Scarborough Road, Briarcliff Manor, NY 10510, May, 1990.

Maes 1989
Maes, P., "How to do the Right Thing," ***Connection Science,*** **1,** no. 3, pp. 291-323, 1989.

Mahadevan 1990
Mahadevan, S., and Connell, J., "Automatic Programming of Behavior-based Robots Using Reinforcement Learning," IBM Research Division Report RC 16359, T. J. Watson Research Center, Yorktown Heights, NY 10598, December 7, 1990

Manning 1989
Manning, C. R., "Introduction to Programming Actors in Acore," in Hewitt, C. and Agha, G. (eds.), ***Concurrent Systems for Knowledge Processing,*** MIT Press, Cambridge, MA 1989.

Mitchell 1986
Mitchell, T. M., Keller, R. M., and Kedar-Cabelli, S. T., "Explanation-based Generalization: A Unifying View," ***Machine Learning,*** **1, 47-80, 1986.**

Mitchell 1990
Mitchell, T. M., "Becoming Increasingly Reactive," *Proc.* AAAZ-90, pp. 1051-1058, July, 1990.

Nilsson 1980
Nilsson, ***N.*** J., ***Principles of Artificial Intelligence,*** Morgan Kaufmann, San Mateo, CA 1980.

Nilsson 1984
Nilsson, ***N.*** (ed.), ***Shakey the Robot,*** SRI Technical Note 323, April 1984, Menlo Park, CA 94025.

Nilsson 1989
Nilsson, N., "Action Networks," ***Proceedings from the Rochester Planning Workshop: From Formal Systems to Practical Systems,"*** Tenenberg, J, et al. (eds.), University of Rochester, Computer Science Technical Report 284, April, 1989.

Nilsson 1990a
Nilsson, N. J., Moore, R., and Torrance, M. C., "ACTNET: An Action Network Language and its Interpreter (A Preliminary Report)," unpublished Stanford Computer Science Dept. Memo, February 22, 1990.

Nilsson 1990b
Nilsson, N. J., "A Scanning Algorithm for Triangle-Table Trees," Unpublished memo, Stanford Computer Science Dept., August 23, 1990.

Pneuli 1989
Pneuli, A., and Rosner, R., "On the Synthesis of an Asynchronous Reactive Module," *Proc.* ***10th Int'l Colloq. on Automata, Languages, and Programming, 372,*** pp. 652-671, Lecture Notes in Computer Science Series, Springer-Verlag, Berlin, July 1, 1989.

Ramadge 1989
Ramadge, P. J. G., and Wonham, W. M., "The Control of Discrete Event Systems," ***Proceedings of the IEEE, 77,*** no. 1, January 1989, pp. 81-98.

Reisig 1985
Reisig, W., ***Petri Nets: An Introduction,*** Springer Verlag, 1985.

Rosenschein 1986
Rosenschein, S. J. and Kaelbling, L.P., "The Synthesis of Machines with Provable Epistemic Properties," In J. F. Halpern, editor, ***Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning about Knowledge,*** pp. 83-98, Morgan Kaufmann Publishers, Inc., Los Altos, California, March 1986. (Updated version: ***Technical Note 412,*** Artificial Intelligence Center, SRI International, Menlo Park, California.)

Schoppers 1987
Schoppers, M. J., "Universal Plans for Reactive Robots in Unpredictable Domains," ***Proceedings of IJCAI-87, 1987.***

Shen 1989
Shen, Wei-Min, ***Learning from the Environment Based on Percepts and Actions,*** PhD thesis, CMU-CS-89-184, Carnegie-Mellon University, School of Computer Science, 1989.

Sutton 1990
Sutton, R. S., "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming," ***Proc Seventh Int'l Conf. on Machine Learning,*** Morgan Kaufmann, San Mateo, CA, June 1990.

Teo 1991
Teo, P., "Botworld," Stanford University Robotics Laboratory, Unpublished Memo, December 10, 1991.

Vere 1978
Vere, S. A., "Inductive Learning of Relational Productions," in ***Pattern-Directed Inference Systems,*** Waterman, D. A. and Hayes-Roth, F. (eds.), pp. 281-295, Academic Press, New York, 1978.

Waterman 1978
Waterman, D. A. and Hayes-Roth, F., "An Overview of Pattern-Directed Inference Systems," in ***Pattern-Directed Inference Systems,*** Waterman, D. A. and Hayes-Roth, F. (eds.), pp. 3-22, Academic Press, New York, 1978.