# 1

# Reacting, Planning, and Learning in an Autonomous Agent

Scott Benson and Nils J. Nilsson

Robotics Laboratory, Department of Computer Science, Stanford University, Stanford, CA 94305

**Abstract**

We present an autonomous agent architecture and its component subsystems that integrate important abilities needed for robust, flexible performance in dynamic environments. These abilities involve appropriate reaction to environmental situations given the agent's goals; selective attention to multiple, competing goals; planning new action routines when innovation beyond designer-provided routines is necessary; and learning the effects of actions so that the planner can use them to build ever more reliable plans. The teleo-reactive format allows actions to be closely coupled to continuous environmental feedback and is also especially compatible with conventional AI planning and learning mechanisms. The workings of the proposed architecture and its subsystems are illustrated in a simulated robot domain. We conclude by noting areas where future work is needed.

## 1 INTRODUCTION

This work introduces and integrates a number of techniques that we think will be important in the development of autonomous agents. By *autonomous*, we mean ones that can operate with only minimal supervision by a human. Such agents would be *managed* by humans, but their actions would not need to be controlled step-by-step by humans. By *agents* we mean computer

systems that perceive and model their environments and take actions in those environments to achieve and maintain goals. Autonomous agents can exist and function in either physical or computational environments. Robots, for example, are physical agents. They sense and act in the physical world. Agents confined to a computational world are sometimes called *knowbots* or *softbots*. These might be simulations of physical robots, information-gathering assistants, or actors in an interactive, virtual reality program. Of course, combinations of both types of agents are possible. Our emphasis here is on robots and simulated robots.

Mobile robots have several important societal applications. Some of these involve manufacturing and warehouse jobs, delivery, transportation, construction, maintenance, janitorial and household tasks, military and space applications, and work in dangerous environments. Such robots must be robust under a wide variety of environmental conditions and be able to survive without human help for long periods of time. The environments in which these robots must operate are dynamic and somewhat unpredictable, containing humans, machines, and other robots.

Here, we focus on four sets of general abilities that we think will be necessary for robots in these applications:

- **Teleo-Reactive Behavior**. We want robots that can react appropriately and rapidly to commonly occurring situations that require stereotypical programs of actions. They must avoid crashing into things, get out of the way of other robots, and refuel themselves, to name just a few examples. But we also want their actions to be influenced by their goals (hence "teleo"). Teleo-reactivity in dynamic, uncertain environments implies a short sense-act cycle of the sort common in feedback control systems. The *teleo-reactive (TR) program* formalism proposed earlier in [Nilsson 1994] seems well suited to these needs.
- **Attention to Multiple Goals**. Our robots will typically have several goals of different and time-varying urgencies. The robots should take actions which, in so far as possible, work simultaneously toward many of their goals—taking

into account the current urgency of each goal and the estimated effort required to achieve it. We have developed an attention-focusing mechanism that is able to adjudicate efficiently among competing goals.

- **Planning**. It will not be possible to store stereotypical action programs matched to all possible situation-goal pairs. Occasional innovation of new action programs and modifications of existing ones will be required. Our agent architecture incorporates a planning system to augment the teleo-reactive programs as needed.
- **Learning**. Constructing a robot control program will require, to be sure, initial human design and coding as well as automatic synthesis by a planning system. Success in complex environments may even require some human recoding of the TR programs. Yet, recoding will be expensive and impractical in many applications, so we seek to incorporate learning and adaptation methods that enable the robot to change its program automatically according to a variety of protocols including declarative instructions from humans, successful and unsuccessful experiences, and supervised and unsupervised training methods.

(In robotics, providing these abilities is usually overshadowed by work on perception and effectors. We have not addressed these critical two components in our research since they depend heavily on the specific application.)

Most of our experimental work on autonomous agents has been carried out using simulated robots and a simulated environment called *Botworld* [Teo 1992]. Botworld is a two-dimensional space of robots, obstacles, bars, and assemblies of bars. Each robot can pick up bars, move them around the environment, and connect them to other bars and structures. Robots can move over bars lying on the "floor" but cannot move through other robots or obstacles.

Each Botworld robot senses the environment at a rapid sampling rate, receiving information about its own position and orientation and whether or not it is holding a bar. It also senses the locations and orientations of nearby bars, obstacles, and

robots. We can simulate imperfect sensors by adding noise to these measurements. A sample scene from Botworld is shown in Fig. 1.1.

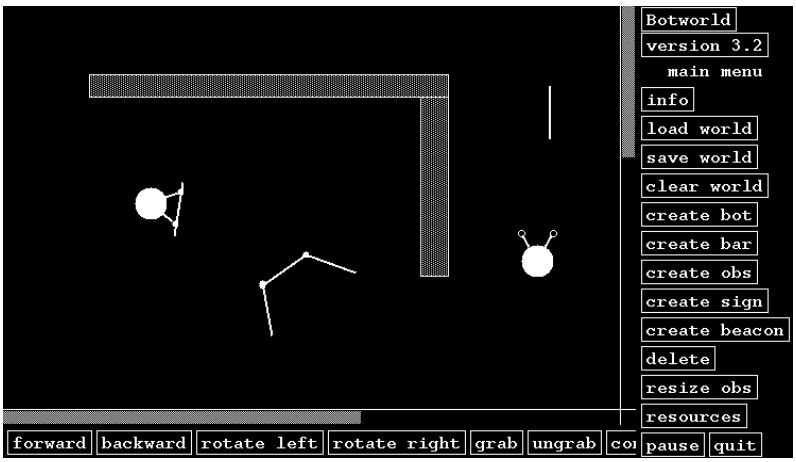


Figure 1.1. Botworld

The next four sections describe the results of preliminary efforts on a coherent architecture and its component subsystems addressing the four abilities listed above. We first summarize the TR program formalism and describe some recent augmentations to it. Next, we describe our architecture for arbitrating among multiple goals within the TR program framework. Then, we present a straightforward, STRIPS-style planning system that is able to create and modify TR programs. Next, we present a learning system that creates models of actions that can be used by the planning system. Along the way, we relate our work to research of others. In the final section, we conclude by noting areas where future work is needed.

## 2 TELEO-REACTIVE BEHAVIOR

### 2.1 Summary of TR programs

A *teleo-reactive (T-R)* program is an agent control program that directs the agent toward a goal in a manner that continuously takes into account changing environmental circumstances. The formalism is described in detail in [Nilsson 1994]. In its simplest

form, a TR program consists of an ordered list of production rules:

$$\begin{array}{rcl} K_1 & \rightarrow & a_1 \\ K_2 & \rightarrow & a_2 \\ & \cdots & \\ K_i & \rightarrow & a_i \\ & \cdots & \\ K_m & \rightarrow & a_m \end{array}$$

The $K_i$ are conditions (on perceptual inputs and on a stored model of the world), and the $a_i$ are actions (on the world or which change the model). An action may be primitive or it may itself be a TR program (thus recursive TR programs are also possible). In typical usage, the condition $K_1$ is a goal condition, which is what the program is designed to achieve, and the action $a_1$ is the null action. The conditions $K_i$ may have free variables which are bound when the TR program is called to achieve a particular ground instance of $K_1$. These bindings are then applied to all the free variables in the other conditions and actions in the program. A TR program is interpreted in a manner roughly similar to the way in which ordered production systems are interpreted: The list of rules is scanned from the top for the first rule whose condition part is satisfied, and the corresponding action is then executed. A TR program is designed so that for each rule $K_i \rightarrow a_i$, $K_i$ is the regression, through action $a_i$, of some particular condition higher in the list. That is, $K_i$ is the weakest condition such that the execution of action $a_i$ (under ordinary circumstances)[1] achieves some particular condition, say $K_j$, higher in the list (with $j < i$). Thus, executing the actions prescribed by a TR program ultimately achieves the goal. Additionally, TR programs are robust in that should an action have an unexpected effect, the program will nevertheless continue working toward the goal.

[1]We assume that an action achieves its expected effects unless some unusual execution error occurs or unless some other agent interferes.

TR programs differ substantively from conventional production systems, however, in that their actions can be *durative* rather than discrete. A durative action is one that can continue indefinitely. For example, a mobile robot is capable of executing the durative action *move*, which propels the robot ahead (say at constant speed) indefinitely. Such an action contrasts with a discrete one, such as *move forward one meter*. In a TR program, a durative action continues so long as its corresponding condition remains the highest true condition in the list. When the highest true condition changes, the action changes correspondingly. Thus, unlike ordinary production systems, the conditions must be *continuously* evaluated; the action associated with the *currently* highest true condition is always the one being executed. An action terminates only when its associated condition ceases to be the highest true condition. The regression condition for TR programs must therefore be rephrased for durative actions: For each rule $K_i \rightarrow a_i$, $K_i$ is the weakest condition such that continuous execution of the action $a_i$ (under ordinary circumstances) eventually achieves some particular condition, say $K_j$, with $j < i$. (The fact that $K_i$ is the *weakest* such condition implies that, under ordinary circumstances, it remains true until $K_j$ is achieved.)

In thinking about the semantics of TR programs, it is important to imagine that the conditions, $K_i$, and all of their parameters, are being *continuously* computed. However, in computational implementations of TR programs, we compute the conditions (and the parameters upon which they depend) at discrete time steps, and then execute small increments of durative actions. A sufficiently high sampling rate is chosen—depending on the domain—to approximate continuous computation and execution.

One way to approximate continuous computation is to evaluate at every time step a Lisp version of a TR program: `(cond (K1 a1) (K2 a2) . . . (Kn an))`. When some of the actions themselves are TR programs, the top-level `cond` statement embeds nested `cond` statements. The program is evaluated at every time step down to a primitive action, an increment of that

primitive action is executed, and program evaluation starts over from the top on the next time step.

This sort of approximation to continuous computation opens the possibility that a condition might be achieved between time steps without being noticed. For instance, if a condition in a TR program is satisfied if and only if the agent is facing within 0.1 degrees of a specific desired direction, the agent might turn sufficiently fast that this condition would hold only *between* two sampling points—leading the agent to turn past the desired direction rather than switching to the appropriate next action. There are at least two obvious solutions to this problem. First, the condition can be relaxed so that at least one sampling point must fall within it, given the agent's execution speed and sampling rate. Second, some measure of the time needed to satisfy the expected next condition can be used to insure that the agent slows down as that condition is approached. This solution has the effect of decreasing the distance between sampling points when necessary so that again, at least one sampling point will necessarily fall within the condition region. Both methods have been implemented successfully for various agent tasks, although the first method occasionally results in activation conditions so weak that the recommended action is no longer always appropriate.

In our work, we have found it convenient to represent a TR program as a tree, called a *TR tree*. Suppose two rules in a TR program are $K_i \rightarrow a_i$ and $K_j \rightarrow a_j$ with $j < i$ and with $K_i$ the regression of $K_j$ through action $a_i$. Then we have nodes in the TR tree corresponding to $K_i$ and $K_j$ and an arc labeled by $a_i$ directed from $K_i$ to $K_j$. That is, when $K_i$ is the shallowest true node in the tree, execution of its corresponding action, $a_i$, will achieve $K_j$. The root node is labeled with the goal condition and is called the *goal node*. When two or more nodes have the same parent, there are correspondingly two or more ways in which to achieve the parent's condition. Continuous execution of a TR tree would be achieved by a continuous computation of the shallowest true node and by execution of its corresponding

action.[2] We call the shallowest true node in a TR tree the *active node*.

We now have had a great deal of experience in writing TR programs for the control of actual physical robots [Galles 1993], the simulated robots of Botworld, and for the control of a Silicon Graphics Flight Simulator aircraft. In the hierarchy of robot control, we have found TR programs to be most appropriate for what might be called *mid-level* robot control. At the lowest level, classical control theory is required for the feedback control of motors and other effectors. Since there is less demand for continuous feedback at the highest levels, conventional program control structures suffice there.

The TR formalism is related to a number of other "circuit-based" agent control methods such as the subsumption architecture [Brooks 1986], universal plans [Schoppers 1987], and situated automata [Rosenschein and Kaelbling 1986]. Comparisons are discussed in [Nilsson 1994]. We prefer the TR formalism because, as we shall see, it is more readily incorporated in an architecture that accomodates planning and learning.

### 2.2 Extensions to the TR Formalism

The basic TR tree formalism has been extended to deal more efficiently with rules having certain kinds of conjunctive conditions. The extension permits subprograms to be executed in arbitrary order depending on circumstances at execution time. It also avoids the necessity of explicit coding of all possible orderings of the subprograms. Consider a condition, $K$, in a TR rule that is composed of a conjunction of subconditions: $K \equiv C_1 \wedge C_2 \wedge \cdots \wedge C_m$. Suppose also, that there are TR programs for achieving each of these subconditions such that if any one of them, say $C_i$, is satisfied, the achievement of any other of them (by their corresponding TR programs) will not cause $C_i$ to become false. That is, the TR programs for achieving the $C_i$ can be executed in any order. In this special case, the TR node labeled by the conjunction is called a *conjunctive node* and is

[2] We assume that ties among equally shallow true nodes are broken by some fixed tie-breaking rule.

given $m$ successor nodes, called *AND nodes*, in the tree—each labeled by one of the conjuncts, $C_i$. (By convention, we do not label the arcs between the AND nodes and their parent conjunctive node.) Each AND node is the root of a TR subtree that achieves its condition without interacting with the other conditions in the conjunction, and thus the TR subtrees can be executed in whatever order circumstances dictate. A TR tree containing AND nodes can be executed by the usual TR execution mechanism: we execute that action corresponding to the shallowest true node, ignoring the AND nodes since they have no action labels on the arcs exiting them. When this special case does not obtain (that is, when programs that achieve the separate conjuncts in a conjunction interact), the entire conjunction must be regressed through all of the actions that achieve the separate conjuncts—resulting in a much larger tree.

An example of a TR tree with AND nodes is shown in Fig. 1.2. This tree will cause a Botworld robot to place two bars at specific locations and orientations in the world and then return to a "home base." Since the acts of placing the two bars can successfully be done in either order, the program for positioning each bar becomes an independent subtree. (Note that the goal node in Fig. 1.2 cannot itself be split into non-interacting conjuncts.)

### 2.3 Simulating Continuous Execution

Approximating continuous computation by searching the entire TR tree at each time step becomes impractical for sufficiently large TR programs. Therefore, we have developed a heuristic method of action selection which usually produces the same result and runs in constant time in nearly all cases regardless of the size of the tree.

In this heuristic method, the agent remembers which node $K_i$ (with associated action $a_i$) was active during the previous time step. It expects that in the absence of surprises, $K_i$ will remain active until its parent node $K_j$ (with associated action $a_j$) becomes active. Therefore, so long as $K_i$ remains true while $K_j$ remains false, $a_i$ is selected as the default action. When and if $K_j$ becomes true, $a_j$ will be selected as the default action.

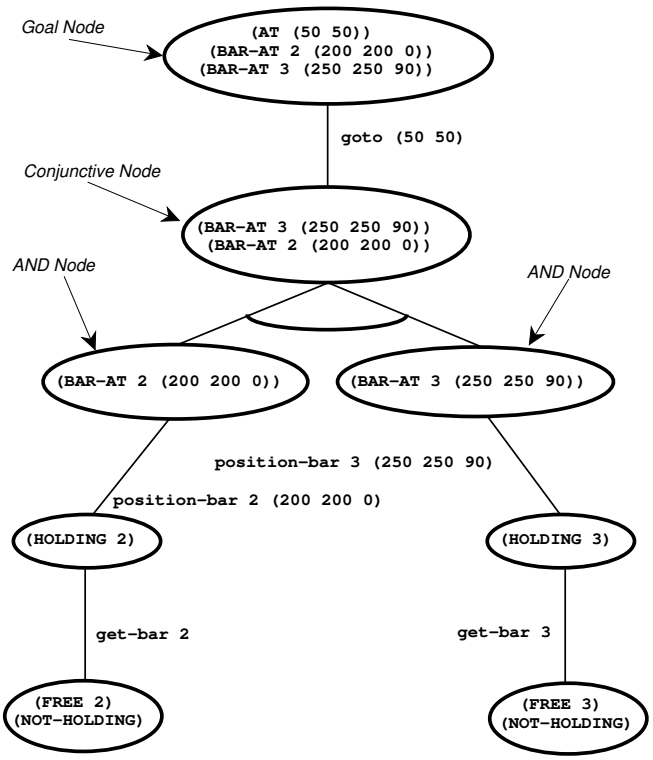


Figure 1.2. A TR Tree with AND Nodes

Only when neither $K_i$ nor $K_j$ holds will the agent fail to have a default action to fall back on.[3]

This default action computation is supplemented by a separate process that scans through the rest of the tree examining a few nodes on each cycle (only nodes higher than the default node need be examined). If this separate process ever finds a true node that is higher than the default node, this higher node will be selected as the active node and the new default. In summary, the agent will execute a normal sequence of goal-achieving actions (so long as they have their expected effects)

[3]Note that only in this case can we not guarantee constant time execution. Since this case arises only due to execution failures, it seems reasonable then to expect a reaction delay. Naturally, we would be forced to provide explict error-handling routines to handle those execution failures in which such reaction delays could not be tolerated.

while searching for serendipitous situations with whatever extra time it has.

## 3 ATTENTION TO MULTIPLE GOALS

In many applications, a robot will have several goals—each presumably achievable by a TR tree. Some goals relate to the maintenance of the robot itself (*e.g.*, when the battery is low, get it recharged), some correspond to low-priority background tasks (*e.g.*, routine mail delivery to offices), and some are given from time-to-time by humans (*e.g.*, deliver this package immediately to room 14). All of these goals must be attended to, although some will be more important than others. The problem of attending to all of them is different than the classical AI problem of conjunctive goals; not all active goals need be satisfied simultaneously. We describe our technique for dealing with multiple goals in the context of our overall agent architecture, illustrated in Fig. 1.3.

Each goal that is being attended to is pursued by a TR program in the TR Memory. These are instantiated from the Plan Library at the time the goal is given to the agent by the user. TR programs corresponding to goals that are to be achieved just once are expunged from the TR memory once their goals are achieved; programs corresponding to goals of maintenance are kept in the TR Memory as long as those goals should be maintained. (Later, we describe the Planner and the Learning System which are used to create new TR programs and to modify existing ones.)

The agent's inputs are such that they allow the computation of conditions needed by its TR programs and by its planning and learning subsystems. The immediate sensory data required for these computations are provided more-or-less continuously by a separate perceptual module—not shown in Fig. 1.3—and are stored in the World Model along with other information about the environment that cannot be immediately sensed but that can be remembered. (The perceptual module is an independent component of the architecture and will not be further discussed here.) Besides having effects on the world, TR programs can

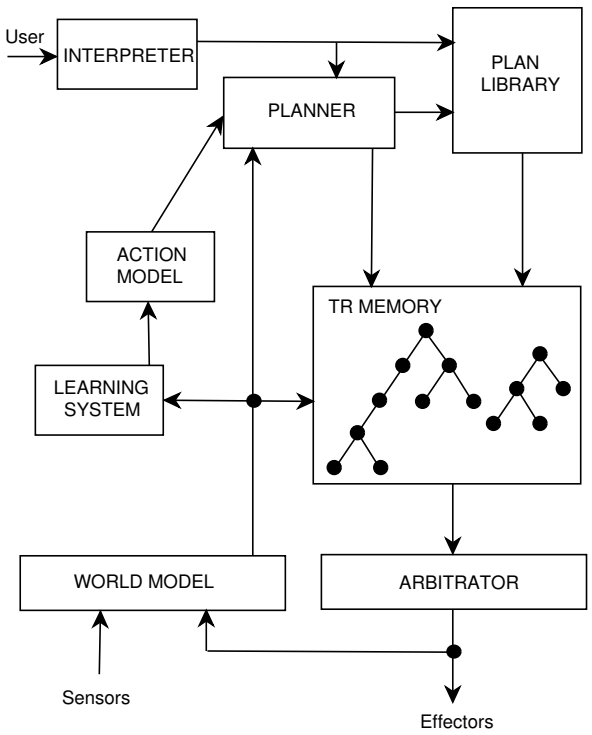


Figure 1.3. Agent Architecture

change the World Model. (Such a change might be appropriate, for example, if a TR program is known to achieve reliably a certain condition that cannot immediately be perceived.)

The Arbitrator decides which of the actions recommended by the separate TR programs in the TR Memory should be selected for execution. For that computation, we have adopted a generalization of the concept of a goal that is similar to that used by researchers in delayed-reinforcement learning [Sutton 1990]. Achieving goal conditions produces *rewards*. Reward amounts can vary according to user priorities and internal and external conditions. Reward possibilities can also evaporate (if the user retracts an assigned task), and therefore the Arbitrator should prefer to work first for high rewards that can be achieved with little effort as soon as possible. This preference is realized by ordering the $m$ goals in the TR Memory in a way that minimizes

$\sum_{i=1}^{m} R_i T_i$, where $R_i$ is the reward for goal $i$ and $T_i$ is the earliest time at which $i$ is expected to be achieved.[4]

A first attempt at optimizing rewards based on this formula might be to consider each goal independently and to select the one which is expected to provide the highest reward per unit time, based on an estimate, $t_i$, of the time that it would take to reach each goal from the *current* situation. At worst, the estimate for $t_i$ can be simply the number of nodes on the path to the root of the TR tree for that goal, but ordinarily the agent will have some better heuristic estimate of the time required by each action.[5] The goal, $i$, which maximizes $R_i/t_i$ then provides the maximum average reward during its completion, and the Arbitrator would select the action corresponding to the shallowest node in the TR tree for goal $i$.

We can state certain conditions under which this greedy strategy is guaranteed to achieve one of the goals in the TR Memory (so long as there are goals to be achieved in the TR Memory):

1. There is an active node in at least one TR tree in the TR Memory,
2. The execution of any action in a TR tree always has its expected effect (*i.e.*, the parent node of the previously shallowest true node eventually becomes true),
3. In every TR tree in the TR memory, the time estimate, $t_i$, of a node is always greater than the time estimate of its parent, and the time estimates for all nodes are fixed,
4. The reward values $R_i$ are constant over time.

However, like most greedy algorithms, it is too focused in its behavior and will prefer one isolated high-reward goal to a set of medium-reward goals even though the medium-reward goals might be much easier to achieve now than they would be after achieving the isolated goal. Also, it is unable to take advantage

[4] Minimizing this sum is a rough approximation to the more difficult problem of maximizing discounted future reward. It is also an instance of a "cable-routing" problem—a special case of which has been studied by [Blum, *et al.* 1994].

[5] For instance, the duration of a `MOVE` action can generally be approximated as a function of the distance to be traveled.

of serendipitous situations which do not involve goal completion. For instance, if one of the agent's lesser goals is to deliver a certain library book to a a far-away office, it will not realize that it can save time by picking up the book while it happens to be in the library for some other reason.

A modification that handles these problems uses the concept of a *stable node*. A condition is stable relative to a set of TR trees if once it holds, it will continue to hold while any of the TR trees in the set are executed. For example, the condition that the agent has a certain book is stable with respect to most TR programs (assuming that the agent has an unlimited carrying capacity), whereas the condition that the agent is in the library is not stable with respect to any TR program that might require the agent to leave the library. By extension, we call a node in a TR tree stable, relative to a set of TR trees, if its associated condition is stable, relative to those TR trees.

Stable nodes can easily be detected using STRIPS-style delete lists. If the condition at a node is not deleted by any of the actions used by a set of TR trees, then that node is stable with respect to those TR trees. We use this process to determine all of the stable nodes in the trees in the TR Memory (with respect to those trees). Since a node is stable or unstable with respect to some context of other TR programs, this process needs to be repeated only when a tree is added to or deleted from the TR Memory.

The Arbitrator uses stable nodes as milestones along the way toward achieving the root node of a TR tree. Instead of computing $R_i/t_i$ for each TR tree in the TR Memory (recall that $t_i$ is the estimate of the time required to achieve the root node of the tree), we compute $R_i/t_i^{sn}$ for each tree, where $t_i^{sn}$ is the estimated time required to achieve the closest stable node in the tree. The Arbitrator selects the action recommended by the TR tree yielding the largest of these quantities. (If a tree has no stable nodes other than the root, we simply compute $R_i/t_i$ for the tree.) Acting to achieve the best stable-node milestones (as measured by $R_i/t_i^{sn}$) helps to ensure that progress already made toward goals is not lost by subsequent actions.

Let us return to consider the situation in which an agent is in the library and has two goals. One is very urgent but involves leaving the library. The other involves delivering a library book to a far-away office. We see that if having the book is a stable node in the program for delivering the book, that delivery task may well have a high value of $R_i/t_i^{sn}$ because $t_i^{sn}$ is small when the agent is in the library. The agent will then pick up the book before working on the other more urgent goal.

Our arbitration method interacts nicely with the TR formalism to select opportunistic actions. For example, suppose the robot always had in its TR Memory the goal of having its battery fully charged. We can assume that the reward value for achieving this goal varies in some manner inversely with the current charge level. Although this goal might not be pursued in a situation in which the battery was reasonably well charged relative to the distance to the charging station, it might happen that on the way to achieving some more important goal the robot happened to pass by the charging station. In this case, depending on its charge level at the time, the Arbitrator might decide to make a small detour for a recharge.

We have integrated this ability to pursue multiple goals into our agent architecture and have verified that it chooses intuitively reasonable actions in a variety of Botworld situations.

Many researchers have proposed architectures for a reactive intelligent agent based on various control schemes. We mention briefly a few representative examples:

- Maes uses spreading activation in behavior networks [Maes 1989] to select an appropriate action based on the agent's goals and percepts. However, behavior networks do not provide any form of hierarchy, allow only discrete actions, and provide no guarantee against looping even in benign situations.
- The BB1 architecture has been used to arbitrate among modules which do planning, goal selection, and execution [Hayes-Roth, *et al.* 1993] for Nomad™ robots. These robots have more high-level capabilities, including meta-planning and motion planning, than is currently supported

by our architecture. However, the BB1 decision cycle does not provide the real-time responses of TR trees, and the present BB1-based architecture can only handle interactions among goals by first merging them and then generating a single plan.

- The subsumption architecture [Brooks 1986] has been used to control a variety of mobile robots that act effectively in the world. However, it is primarily concerned with lower-level issues than those addressed by our architecture; it does not presently have the capability to arbitrate among multiple varying goals or to plan for novel goals.
- Mitchell has proposed a system called the Theo-Agent [Mitchell 1990] which combines planning, learning, and reaction through gradual compilation of stimulus-response rules. The Theo-Agent's performance improvement through speedup learning is beyond the current capabilities of our TR agent. However, the Theo-Agent does not perform inductive learning, or goal-arbitration.
- A variety of agents have been created in SOAR [Laird and Rosenbloom 1990, Jones, *et al.* 1993], including a mobile robot and a flight simulator pilot. The SOAR architecture [Laird, *et al.* 1987] provides a unified framework for an agent architecture, as well as speed-up learning and planning systems. However, none of the SOAR agents appear to handle multiple, time-varying goals.
- Gat has implemented a control architecture called ATLANTIS which integrates reactive behavior with a motion planning system [Gat 1992]. The system does not yet have a learning component nor a goal arbitrator.
- The PRS system [Georgeff and Lansky 1987] achieves both reactivity and goal-directedness by combining a reasoning system and a library of plans covering commonly occuring tasks. PRS is probably the most similar architecture to ours, but it does not incorporate learning.

## 4 PLANNING

We turn now to the matter of creating new TR programs and modifying existing ones by an automatic planning system. TR programs resemble the search trees constructed by backward-chaining AI planning systems. The overall goal is the root of the tree; any non-root node $i$ is the regression of its parent node, $j$, through the action, $a_i$, connecting them. We have exploited this similarity by developing an automatic planning system that regresses conditions through durative actions to build a search tree. The search tree is then converted in a straightforward manner to a TR program.

Our planner uses STRIPS-like operators [Fikes, *et al.* 1972] to model the effects of durative or atomic actions. A durative action itself does not have a STRIPS-like model, though, because a durative action has several effects depending on how long it is executed. STRIPS operators assume that actions are discrete with definite effects.

The representation we use, which we call a *teleo operator (TOP)*, is based on the backward regression of a literal through an action. The literals that we use are all those that we might want as components of TR conditions. For any literal $\lambda_i$ and any action $a_j$, we can define a TOP, $\tau_{ij}$, that describes the process of continuously executing $a_j$ until $\lambda_i$ becomes true. (If executing $a_j$ never makes $\lambda_i$ true, there is no such TOP.) The planner can treat a TOP model of a durative action as if the action were discrete. When the planner calls for the execution of an action corresponding to a TOP, it intends that that action be executed just exactly as long (no longer, no shorter) as is needed to achieve its intended effect. Thus a TOP describes a discrete action and can be used exactly like a STRIPS operator.

Following work on directional preimages in robot motion planning [Lozano-Pérez *et al.* 1984] we define the *preimage*, $\pi$, of a literal, $\lambda$, through durative action, $\alpha$, as the weakest condition under which continuous execution of $\alpha$ will eventually satisfy $\lambda$ while maintaining $\pi$ until $\lambda$ does becomes true. In analogy with STRIPS operators, $\pi$ is the precondition for the TOP that models $\alpha$ when used to achieve $\lambda$.

Since we are defining TOPs only for each literal-action pair, when the planner regresses a more complex condition through a TOP, it will need the equivalent of STRIPS add and delete lists, so our TOPs will have these also. In our system, a TOP, then, has four components. These are:

1. the name of the action, $a$
2. a *postcondition* literal, $\lambda$, that is the intended effect of the TOP
3. a *preimage*, $\pi$, of the TOP
4. the set of *side effects*, $S$, of the TOP

Components of the side effects are analogous to add and delete lists. For a TOP with action $a$, postcondition $\lambda$, and preimage $\pi$, a side effect is a literal, $\sigma$, which is not necessarily true at the time action $a$ was begun but (usually) becomes true by the time $\lambda$ becomes true. Positive-literal side effects correspond roughly to elements of a STRIPS add list, and negative-literal side effects correspond to elements of a STRIPS delete list. Given the set of side effects, the planner can now do regression on a condition of any form using the standard STRIPS regression methods—treating the positive and negative side effects as the add and delete lists of the TOP, respectively.

Our planner can generate hierarchical and nonlinear plans, and can extend an existing plan to handle unexpected circumstances. Given any goal and a current situation, the planner does breadth-first backward-chaining until one or more subgoals are produced that hold in the current situation. The paths from all currently satisfied nodes to the goal are included in a TR tree for this goal, while the unsuccessful branches of the search may either be retained, potentially increasing the coverage of the tree, or pruned as likely to be irrelevant. Currently, unsuccessful branches are pruned to prevent overly large trees.

Hierarchical planning is implemented by having the planner develop TR programs for achieving a set of designer-specified goals, such as `(HOLDING ?x)`. These programs are given names, such as `(pickup-bar)`, generalized (by replacing constants by variables as appropriate [Fikes, *et al.* 1972]), and stored as macro-operators (with their effects described by TOPs). The

planner can then utilize these macro-operators as actions when constructing trees for other goals. An example of a hierarchical Botworld plan constructed by the system is shown in Fig. 1.4.

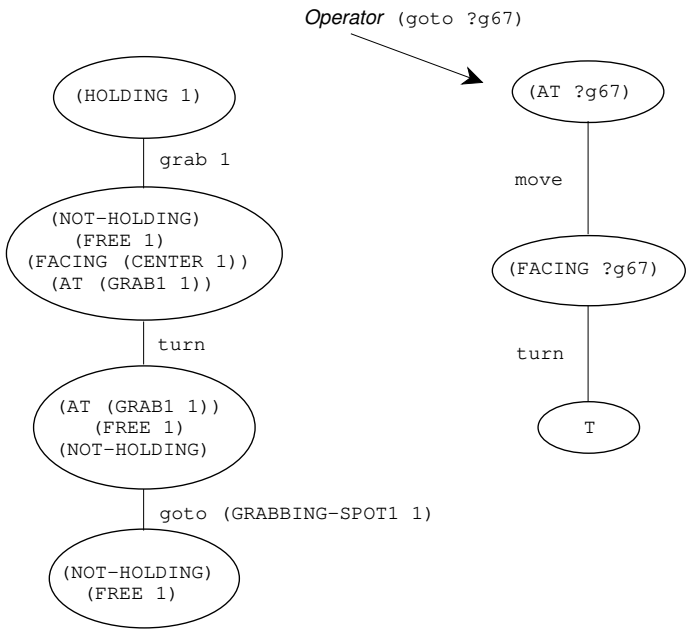


Figure 1.4. A Hierarchy of TR Trees Produced by the Planner

Only a simplified form of nonlinear planning is implemented at present. (See [McAllister and Rosenblitt 1991] for a thorough discussion of nonlinear planning.) When faced with any conjunctive goal or subgoal, our planner first attempts to treat each conjunct as an independent subgoal, creating AND node successors for each of them. If a conjunctive goal cannot be split in this way (because achieving one of its conjuncts would render an already achieved one false), the conjunction is treated instead as a single goal that must be regressed in its entirety through TOPs to produce successors.

A key advantage of the TR tree formalism over other reactive formalisms is the ease of replanning during execution [Nilsson 1994]. Since the initial trees generated by the planner do not usually represent universal plans, situations will arise

during execution that are not covered by any node in the current tree. (Such a circumstance can occur for a variety of reasons, including probabilistic action outcomes, execution errors, and the actions of other agents.) Instead of requiring complete replanning, these unexpected situations will usually require only an extension of the existing TR program. Such an extension can be made by having the planner backward chain from each node in the current TR program until the novel condition is reached. Since the existing nodes in a TR tree are retained in this process, the coverage of the tree increases monotonically as the agent gains more experience in the world. The revised trees are kept in the Plan Library for future use, and thus the agent's TR programs cover an increasing variety of situations that are representative of the situations actually encountered. Of course, pausing for plan extension causes a delay before action can be resumed, but we assume that such planning occurs asynchronously with continuous monitoring of the trees in the TR Memory. If situations requiring action occur (recognizable by the existing TR trees), the appropriate actions will be taken.

We have incorporated this planning system into our agent architecture, and have demonstrated smooth interactions between the execution of existing TR programs and their modifications by the planner.

## 5 LEARNING

### 5.1 A Method for Learning TOPs

The ability to learn from experience and from instruction is a necessary component of useful robots. We are exploring a number of learning mechanisms in our research. Perhaps the simplest of these is the ability to learn categories of physical locations from sensor data. In [Galles 1993], we endowed a mobile Nomad$^{TM}$ robot with a neural-net-like structure with sonar inputs to learn how to distinguish between different locations (such as corridors, $\top$-junctions, and corners) which it encountered in a simple office environment. The system represents these locations in a topological map that it makes of this environment. In other work, John has proposed and imple-

mented a method for direct supervised learning of TR programs [John 1993].

Several authors have used reinforcement learning techniques to learn action policies that lead to rewards [Mahadevan and Connell 1992, Sutton 1990]. One of the problems with reinforcement learning methods is that the policies learned depend too much on the reward structure used during learning. We would like what is learned while pursuing one set of goals to transfer easily to another set whenever possible. Therefore, we have concentrated on learning TOP models of the effects of actions and then using these models in a planner to produce TR programs.

We have developed a technique for learning and updating TOPs from both directed and random exploration of an environment by a robot. We have integrated this style of learning into our architecture and have demonstrated that it does indeed produce TOPs that can be used by our planning system.

To learn a TOP, we first decide on a set, $\Gamma$, of predicates thought to be needed to describe adequately the state of the environment and the effects of actions (postconditions, preimages, and side effects). These predicates must be ones that are continuously computable from perceptual and stored information. Learning is based on experiences recorded during one or more training runs. Each run consists of a series of actions taken in the world, whether guided by a teacher, resulting from random exploration, or produced during problem solving. While the agent is acting and perceiving during training, the values of all predicates in $\Gamma$ are collected at every time step. At the $i$-th time step, these values are represented by a *state formula*, $\xi_i$. (For example, if the predicates in $\Gamma$ are $x_1$, $x_2$, and $x_3$ with values at time step $i$ of $T$, $T$, and $F$, respectively, the corresponding $\xi_i$ would be $x_1 \wedge x_2 \wedge \neg x_3$.) We first describe how these state formulas are used to learn the preimages of TOPs.

Recall from our earlier description of TOPs that if a preimage, $\pi$, of a TOP, $\tau_{ij}$, is entailed by a formula, $\xi$, then the durative execution of action $a_j$ starting in an environmental state characterized by $\xi$ is predicted eventually to lead to a state characterized by a formula that entails the postcondition $\lambda_i$. Thus,

learning a preimage is a concept learning problem in which we seek to induce a formula $\pi$ that is entailed by all of the "positive" state formulas and is not entailed by any of the "negative" state formulas. Positive examples for the TOP, $\tau_{ij}$, are provided by those training instances in which the durative execution of $a_j$ actually did lead to $\lambda_i$, and negative ones are provided by those instances that did not lead to $\lambda_i$. For the moment, let's postpone the details of how we partition the state formulas collected during the training runs into positive and negative sets. For now, we assume we have a set $\Xi^+$ of positive formulas and a set $\Xi^-$ of negative formulas for each TOP to be learned. From these, we use a simple concept learning method that finds a disjunctive-normal-form formula that is entailed by all of the formulas in $\Xi^+$ and is not entailed by any of the formulas in $\Xi^-$.

We now describe the method for learning a particular TOP, $\tau_{ij}$. (TOPs are learned for each action $a_j$ and each literal, $\lambda_i$ composed of predicates in $\Gamma$.)

- Set the initial preimage condition, $\pi$, of the TOP to $F$ (*False*).
- Iterate over each (positive) $\xi_i \in \Xi^+$ (this outer loop creates disjunctions):
    - If $\pi$ subsumes $\xi_i$ (*i.e.*, if $\xi_i \models \pi$), go on to the next example in $\Xi^+$.
    - Otherwise, create a new concept $\phi_i \leftarrow \xi_i$.
    - Iterate through the other positive examples $\{\xi_j \mid j \neq i\}$ (this inner loop creates conjunctions of literals):
        - compute $C(\phi_i, \xi_j)$, where $C(\alpha, \beta)$ is the *consensus* of $\alpha$ and $\beta$. (The *consensus* of two conjunctions, $\alpha$ and $\beta$, is the conjunction of all and only those literals appearing in either $\alpha$ or $\beta$ that do not appear positively in one and negatively in the other. For example, the consensus of $x_1 \wedge \neg x_2 \wedge x_3$ and $x_2 \wedge x_3$ is $x_1 \wedge x_3$. The consensus of $\alpha$ and $F$ is $\alpha$. The consensus of $\alpha$ and $T$ is $T$.)

- If $C(\phi_i, \xi_j)$ does not subsume any (negative) $\xi \in \Xi^-$, set $\phi_i \leftarrow C(\phi_i, \xi_j)$. (This operation generalizes $\phi$ to subsume other members of $\Xi^+$ when such a generalization doesn't include any negative examples. That is, $\phi_i$ is the most specific conjunction that includes $\xi_i$ and as many of the other $\xi$ in $\Xi^+$ as possible without including any negative examples.)
- Set $\pi \leftarrow \pi \vee \phi_i$.

We next describe how we obtain positive and negative state formulas for learning the preimage of $\tau_{ij}$. We first find a trial preimage from the positive examples, and then use the trial preimage to identify negative examples.

- All state formulas, $\xi_i$ (with $i$ now an index over time steps), collected during a training run are assembled in a *sequence trace*, $\Sigma$.
- We first collect positive state formulas for learning the preimage of $\tau_{ij}$ by isolating all subsequences of $\Sigma$ during which action $a_j$ is being continuously executed and for which $\lambda_i$ is false during the entire subsequence but then becomes true at the end of the subsequence. For example, suppose one such subsequence, $\Sigma_{ij}$, is $\{\xi_1, \xi_2, \ldots, \xi_i, \ldots, \xi_k\}$, where $\lambda_i$ is true only in the last formula, $\xi_k$. Each of $\xi_1, \ldots \xi_{k-1}$ is a positive instance of the preimage $\pi_{ij}$.
- Next, we apply the above concept learning algorithm using the positive instances only. (This application will typically result in an overly general preimage.)
- We use this possibly over-general preimage, $\pi$, to identify a set of negative examples by isolating all subsequences of $\Sigma$ during which action $a_j$ is being continuously executed and during which $\pi$ holds of the formulas in the subsequence. For each such subsequence $\{\xi_1, \xi_2, \ldots, \xi_i, \ldots, \xi_k\}$, in which $\lambda_i$ never holds, the $\xi_i$ are all negative examples if either:
  - $\pi$ no longer holds in $\xi_{k+1}$ while $a_j$ is still being executed, or

* $\lambda_i$ is not achieved during the entire time $a_j$ is being executed, and if the total length of time for which $a_j$ was executed without achieving $\lambda_i$ is significantly greater than the average time $\tau_{ij}$ takes to achieve $\lambda_i$. Currently, "significantly greater" means greater than the average by a factor of at least 3.

Now that we have a set of positive and negative examples, we reapply the learning algorithm to produce a more specialized preimage that excludes the negative examples. Since the addition of negative examples can only make the preimage more specific, there is no need to repeat the process of finding more negative examples.

We compute the side effects of $\tau_{ij}$ by examining the "positive" subsequences, $\Sigma_{ij}$. The side effects of $\tau_{ij}$ are all of those literals that have values in $\xi_k$ (the "final" state formula that entails $\lambda_i$) that are opposite from their values in at least one of $\xi_i$ for $i < k$ in $\Sigma_{ij}$. It is, of course, possible that a literal determined to be a side effect in one subsequence might not be a side effect in another. We attach to each side effect, $\sigma$, a conditional probability estimate, $p$, which measures the proportion of those subsequences for $\tau_{ij}$ in which $\sigma$ is true in $\xi_k$, given that $\sigma$ was false at some earlier $\xi_i$. When using learned TOPs for planning, the planner will consider only side effects with probabilities above some threshold; other side effects correspond to superstitions that arise by chance coincidence. If a side effect $\sigma$ is a positive literal, it is called a positive side effect; otherwise it is called a negative side effect.

This learning procedure may be repeated an arbitrary number of times as the agent gains experience. Each time the agent takes some series of actions in the world, it provides the action model learner an opportunity to refine its models. Thus, the TOPs become increasingly close approximations of the true effects of the actions.

### 5.2 Sample Run

We illustrate our learning method by showing how it learns the effects of the actions used in grabbing a bar in Botworld. The

learned TOPs are then used by the planner to construct a TR program for bar grabbing. We assume that the agent's perceptual system computes whether or not:

- it is holding a bar
- it is facing the bar midline
- it is aligned with the bar midline
- it is at the correct distance from the bar in order to grab it
- it is parallel to the long axis of the bar
- it is facing the center of the bar

These predicates comprise the elements of $\Gamma$ for this illustrative example. Most of the relevant terms used in these predicates are shown in Fig. 1.5.

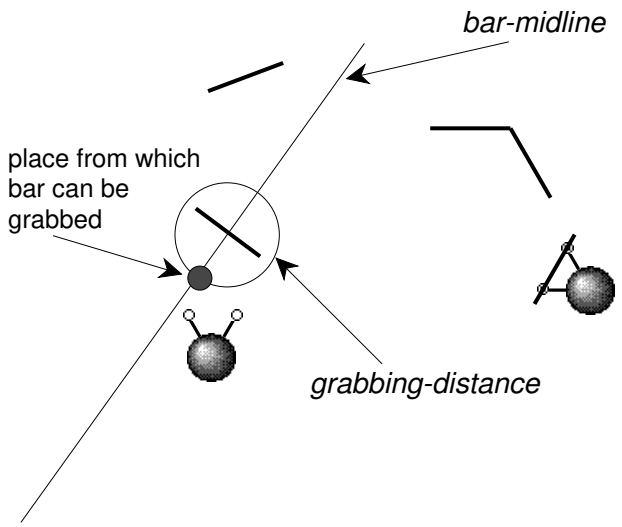


Figure 1.5. Terms Used in Botworld Predicates

Using these predicates, the agent collects a series of experimental traces, some representing successful bar-grabbing runs guided by a human-coded TR program, and some representing random exploration of the world. The learning system then produces TOPs from these experiences. Between 19 and 21 TOPs

<(HOLDING ?x), grab>
Preimage condition: (AT-GRABBING-DISTANCE ?x) ∧
(FACING-BAR ?x) ∧ (ON-MIDLINE ?x)
Average timing: 1.0
Side effects: none

<(AT-GRABBING-DISTANCE ?x), forward>
Preimage condition: (NOT (TOO-FAR ?x)) ∨ (FACING-BAR ?x)
Average timing: 17.0
Side effects: −(TOO-FAR ?x) [100%]
−(ON-MIDLINE ?x) [55%]
−(TOWARD-MIDLINE ?x) [63%]

<(PARALLEL-TO ?x), turn>
Preimage condition: TRUE
Average timing: 25.5
Side effects: −(FACING-BAR ?x) [99%]
+(FACING-BAR ?x) [3%]
+(TOWARD-MIDLINE ?x) [58%]
−(TOWARD-MIDLINE ?x) [45%]

Figure 1.6. Three TOPs Learned During Some Training Runs

were learned on each trial run, of which 7 are essential for the bar grabbing task. The other TOPs represented reasonable but unnecessary operators such as turning to face away from the bar.

The learned TOPs were then given to our TR planner which planned a TR tree to accomplish the bar-grabbing task. This learned/planned tree was successful in completing the task in all of the examples we gave it.

Three of the TOPs learned by the system are shown in Fig. 1.6. A "+" sign in front of a side effect expression indicates it is used as an add-list item; a "−" sign indicates it is used as a delete-list item. Generalization was done by a simple constant-to-variable conversion similar to that done for storing plans in the Plan Library.

All three of the TOPs shown in the figure represent correct and useful operations, but there are several details worth mentioning. First, there are two different situations in which the agent can move forward to achieve `AT-GRABBING-DISTANCE` for a bar, either by being too close to the bar to start with, or by facing the bar directly from any distance. Both situations are represented in the preimage of the TOP <`(AT-GRABBING-DISTANCE ?x), forward`>. Second, the predicate `(PARALLEL-TO ?x)` has an interesting relationship to the predicate `(FACING-BAR ?x)` in that they can be simultaneously true only in the rare circumstance in which the robot is on the long axis of the bar bound to `?x`. This relationship is reflected in the side effects of the TOP <`(PARALLEL-TO ?x), turn`> in that `(FACING-BAR ?x)` has a very high, but not certain, probability of being deleted when `turn` is executed, and a small but non-zero chance of being added when `(PARALLEL-TO ?x)` is achieved.

### 5.3 Related Work on Learning

The TOP model of actions and the method of learning from positive experiences derive from Vere's relational predicate learning algorithm[Vere 1980]. However, all of Vere's learning occured in deterministic and discrete domains. Our TOP models are similar to the low-level goal achievement rules used by the compiler in GAPPS [Kaelbling and Rosenschein 1990]. The GAPPS rules do not include side effects, and are explicitly programmed rather than learned.

TOPs are quite similar to the empirical backprojections in [Christiansen 1991]. Christiansen's work is the only work we have seen that learns action models on a real physical robot. However, Christiansen's algorithm used only discrete actions and operated in a very simple environment.

The problem of learning action models from experience has been studied extensively by [Gil 1992] and by [Shen 1989]. Both of them assumed deterministic domains, and assumed primarily discrete worlds. One of Shen's experiments did involve learning action models in a continuous robot arm domain and had a somewhat more quantitative emphasis than does the work described here.

Mahadevan learns action models for a robot in a continuous domain sensed through a 12 by 12 certainty grid [Mahadevan 1992]. These models are learned for one step actions only and provide no method of focusing on a specific aspect of the environment. Grefenstette and Ramsey view the process of learning action models for a robot as learning parameters of user-defined operators, using an algorithm they call *case-based anytime learning* [Grefenstette and Ramsey 1992]. Their work is a promising approach to robot control, but relies heavily on the user defined parameterized operators and is more quantitative than our approach.

## 6 CONCLUSIONS AND FUTURE WORK

We have presented an agent architecture that integrates the ability to react appropriately in dynamic worlds with the abilities to plan and to learn. Although the work reported here is preliminary and has been tested only on simulated robots, we think the ideas embodied in this work exemplify what will be required for robust, adaptive, and flexible autonomous agents. Our research has also given us some perspective about the extensive work that remains to be done.

We now mention some areas that we intend to pursue and cite some preliminary work by others in each of these. First, we have assumed that the perceptual system is able to compute the values of a set of predicates on which the agent bases its actions and its planning and learning. Providing an adequate perceptual system for a robot requires advances in computer vision and other sensory modalities. In this connection, we will be looking closely at the sort of work being done by [Horswill 1993] and by [Ballard 1991] on "active vision" in which processing is matched to the particular task at hand rather than on complete analyses of scenes.

We have assumed that the agent's designer specifies the sensors and the perceptual predicates that will be needed. But in many applications the designer will not be able to predict all of the predicates that the agent will need—even though the set of sensors on which all perceptual predicates are based will nec-

essarily be fixed by the designer. It will be important for the agent to be able to extend its set of predicates automatically to cover aspects of its world that turn out to be important. Some initial work in this area has been done by [Shen 1989] and by [Drescher 1991].

On the other side of the perception-action spectrum, we have provided a few primitive actions and some higher level routines constructed from these primitives. Because of insurmountable combinatorial problems, both planning and learning require hierarchies of actions. Planning must operate first at high levels of the hierarchy and then articulate these plans in increasing detail. We imagine that the learning system will first construct low level TOPs and then gradually learn the effects of higher level actions. Some work has already been done on learning hierarchies of actions in domains unfamiliar to the learner. Drescher's schema mechanism develops complex actions [Drescher 1991], while Ring combines actions to create operators using reinforcement learning [Ring 1991].

Several other automated planning techniques could be effectively used in conjunction with our architecture. Two promising directions involve "anytime planning" [Boddy and Dean 1989] and temporal reasoning [Dean and Wellman 1991]. Anytime planning would allow the agent to react as best it can in critical situations where there is insufficient time to plan a complete TR tree before acting. Reasoning about time is necessary if the agent must deal with goals that have real-time deadlines.

There are several directions in which the learning component of the architecture could be extended. One problem with the current method is that the total number of TOPs that might be created is the product of twice the number of predicates times the number of actions. While this product was not excessive in the Botworld domain, in more complex environments a large number of TOPs would severly slow any planner trying to use them. Discarding seldom-used TOPs might keep the number of TOPs managable.

Another clear limitation of the present learning system is its dependence on a teacher to generate useful experiences. The TOP learning algorithm does not itself depend on the experi-

ences being goal-directed, but (just as in reinforcement learning) random exploration does not provide sufficient coverage of the state space to learn all of the operators the planner might need. (For example, in Botworld, the likelihood that the robot would ever successfully execute a `GRAB` action during random exploration is extremely remote.) Exploration and experimentation have been studied in other work on learning action models [Gil 1992, Shen 1989], and these studies will provide a starting point for our future work.

We also intend to try more sophisticated concept learning algorithms (*e.g.*, C4.5 [Quinlan 1992] or FOIL [Quinlan 1990]) to approximate the preimage of a TOP.

Constructing a useful World Model can also be considered a problem in machine learning. A key problem here for mobile robots involves building some kind of map of the environment and then using this map first to avoid getting lost and (when needed) to re-orient. Kuipers has worked extensively on map-making [Kuipers 1991], and we have done some preliminary work [Galles 1993], which we plan to extend.

Finally, our ideas and their planned extensions must be tested in other, more complex domains. The Botworld tasks have provided useful preliminary tests, and we are now considering other application domains. Among these are: more work with the Silicon Graphics flight simulator, control of a physical mobile robot in office domains, and the interactive system of "Woggleworld" cartoon characters [Bates 1992].

## 7 ACKNOWLEDGEMENTS

This work was partially supported by the National Science Foundation under grant number IRI-9116399 and by the Advanced Research Projects Agency under grant number F49620-94-1-0090 (monitored by the Air Force Office of Scientific Research). Mr. Benson was partially supported by a National Defense Science and Engineering Graduate Fellowship. The authors are grateful for the useful comments made by Andrew Kosoresow.

## REFERENCES

[Ballard 1991] Ballard, D., "Animate Vision," *Artificial Intelligence*, 48, pp. 57-86, 1991.

[Bates 1992] Bates, J., "Virtual Reality, Art, and Entertainment," *PRESENCE: Teleoperators and Virtual Environments*, 1(1), pp. 133-138, 1992.

[Blum, *et al.* 1994] Blum, A., *et al.*, "The Minimum Latency Problem," *Proc. of STOC-94*, pp. 163-171, Association for Computing Machinery, 1994.

[Boddy and Dean 1989] Boddy, M., and Dean, T., "Solving Time-Dependent Planning Problems," *Proc. IJCAI-89*, pp 979-984, San Francisco: Morgan Kaufmann, 1989.

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

[Christiansen 1991] Christiansen, A., "Manipulation Planning from Empirical Backprojection," *Proc. IEEE Conference on Robotics and Automation*, 1991.

[Dean and Wellman 1991] Dean, T., and Wellman, M., *Planning and Control*, Chapter 3, San Francisco, CA: Morgan Kaufmann, 1991.

[Drescher 1991] Drescher, G., *Made Up Minds: A Constructivist Approach to Artificial Intelligence*, Cambridge, MA: MIT Press, 1991.

[Fikes, *et al.* 1972] Fikes, R., Hart, P., and Nilsson, N.J., "Learning and Executing Generalized Robot Plans," *Artificial Intelligence*, 3, pp. 251-288, 1972.

[Galles 1993] Galles, D. , "Map Building and Following Using Teleo-Reactive Trees," in *Intelligent Autonomous Systems: IAS-3*, Groen, F. C. A., Hirose, S. and Thorpe, C. E. (Eds.), pp. pp. 390-398. Washington: IOS Press,1993.

[Gat 1992] Gat, E., "Integrating Planning and Reacting in a Heterogeneous Asynchronous Architecture for Controlling Real-World Mobile Robots," *Proc. of AAAI-92*, pp. 809-815, Cambridge, MA: MIT Press, 1992.

[Gil 1992] Gil, Y., "Acquiring Domain Knowledge for Planning by Experimentation," PhD Thesis, Carnegie Mellon University, 1992.

[Georgeff and Lansky 1987] Georgeff, M., and Lansky, A., "Reactive Reasoning and Planning," *Proc. of AAAI-87*, pp. 677-682, San Francisco, CA: Morgan Kaufmann, 1987.

[Grefenstette and Ramsey 1992] Grefenstette, J.J., and Ramsey, C.L., "An Approach to Anytime Learning," *Machine Learning: Proceedings of the Ninth International Workshop*, pp 189-195, San Francisco, CA: Morgan Kaufmann, 1992.

[Hayes-Roth, *et al.* 1993] Hayes-Roth, B., Lalanda, P., Morignot, P., Pfleger, K., and Balabanovic, M., "Plans and Behavior in Intelligent Agents," KSL Report No. KSL 93-43, Stanford University, May 1993.

[Horswill 1993] Horswill, I., "Polly: A Vision-Based Artificial Agent," *Proc. of AAAI-93*, pp. 824-829, Cambridge, MA: MIT Press, 1993.

[John 1993] John, G., "SQUISH: A Preprocessing Method for Supervised Learning of T-R Trees from Solution Paths," (unpublished) Robotics Laboratory, Stanford University, 1993.

[Jones, *et al.* 1993] Jones, R.M., Tambe, M., Laird, E., and Rosenbloom, P., "Intelligent Automated Agents for Flight Training Simulators," *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representations*, March, 1993.

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

[Kuipers 1991] Kuipers, B., "A Robot Exploration and Mapping Strategy Based on a Semantic Hierarchy of Spatial Representations," *Robotics and Autonomous Systems*, 8, 1991.

[Laird, *et al.* 1987] Laird, J., Newell, A., and Rosenbloom, P., "SOAR : An Architecture for General Intelligence," *Artificial Intelligence*, 33(1), pp.1-64, September, 1987.

[Laird and Rosenbloom 1990] Laird, J., and Rosenbloom, P., "Integrating Execution, Planning, and Learning in Soar for External Environments," *Proc. of AAAI-90*, Cambridge, MA: MIT Press,1990.

[Lozano-Pérez *et al.* 1984] Lozano-Pérez, T., Mason, M.T., and Taylor, R.H, "Automatic Synthesis of Fine-Motion Strategies for Robots," *International Journal of Robotics Research*, 3(1), pp. 3-24, 1984.

[McAllister and Rosenblitt 1991] McAllister, D. and Rosenblitt, D., "Systematic Nonlinear Planning," *Proc. of AAAI-91*, Cambridge, MA: MIT Press,1991.

[Maes 1989] Maes, P., "How to Do the Right Thing," *Connection Science Journal*, 1(3), pp. 291-323, 1989.

[Maes 1990] Maes, P., "Situated Agents Can Have Goals," *Robotics and Autonomous Systems, 6*, North-Holland, 1990.

[Mahadevan 1992] Mahadevan, S., "Enhancing Transfer in Reinforcement Learning by Building Stochastic Models of Robot Actions," *Machine Learning: Proceedings of the Ninth International Workshop*, San Francisco, CA: Morgan Kaufmann, 1992.

[Mahadevan and Connell 1992] Mahadevan, S., and Connell, J., "Automatic Programming of Behavior-Based Robots Using Reinforcement Learning," *Artificial Intelligence*, 56(2-3), pp. 311-365, 1992.

[Mitchell 1990] Mitchell, T., "Becoming Increasingly Reactive," *Proc. of AAAI-90*, Cambridge, MA: MIT Press, 1990.

[Nilsson 1994] Nilsson, N. J., "Teleo-Reactive Programs for Agent Control," *Journal of Artificial Intelligence Research*, 1, pp.139-158, January, 1994.

[Quinlan 1990] Quinlan, J. R., "Learning Logical Definitions from Relations," *Machine Learning*, 5(3), pp. 239-266, 1990.

[Quinlan 1992] J. Ross Quinlan, *C4.5: Programs for Machine Learning*, San Francisco, CA: Morgan Kaufmann, 1992.

[Ring 1991] Ring, M., "Incremental Development of Complex Behaviors Through Automatic Construction of Sensory-Motor Hierarchies," in Birnbaum, L., and Collins, G. (Eds.), *Machine Learning: Proceedings of the Eighth International Workshop*, pp. 343-347, San Francisco, CA: Morgan Kaufmann, 1991.

[Rosenschein and Kaelbling 1986] Rosenschein, S. J., and Kaelbling, L.P,, "The Synthesis of Machines with Provable Epistemic Properties," in *Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning about Knowledge*, Halpern, J. (Ed.), pp. 83-98, San Francisco, CA: Morgan Kaufmann. (Updated version: Technical Note 412, Artificial Intelligence Center, SRI International, Menlo Park, CA, 1986.)

[Schoppers 1987] Schoppers, M. J., "Universal Plans for Reactive Robots in Unpredictable Domains," in *Proceedings of IJCAI-87*, San Francisco, CA: Morgan Kaufmann, 1987.

[Shen 1989] Shen, W.M., "Learning from the Environment Based on Actions and Percepts," PhD Thesis, Carnegie Mellon University, 1989.

[Sutton 1990] Sutton, R., "Integrated Architectures for Learning, Planning, and Reacting based on Approximating Dynamic Programming," *Machine Learning: Proceedings of the Seventh International Workshop*, pp. 216-224, San Francisco: Morgan Kaufmann, 1990.

[Teo 1992] Teo, P. "Botworld," (unpublished) Robotics Laboratory, Computer Science Dept., Stanford University, December, 1992.

[Vere 1980] Vere, S. A., "Multilevel Counterfactuals for Generalizations of Relational Concepts and Productions," *Artificial Intelligence*, 14, pp.139-164, 1980.