I. Agent Architectures

Can anything in general be said about intelligent agent architectures? [1] Just as there are millions of species of animals, occupying millions of different niches, I expect that there will be many species of artificial agents—each a specialist for one of a countless number of tasks. The exact forms of their architectures will depend on their tasks and their environments. For example, some will work in time-stressed situations in which reactions to unpredictable and changing environmental states must be fast and unequivocal. Others will have the time and the knowledge to predict the effects of future courses of action so that more rational choices can be made. Even though there will probably never be a single, all-purpose agent architecture, there is one that I think might play a prominent role in many future systems. It can be viewed as an elaboration of the first two levels of the popular three-level architectures that have been prominent in robotics research.

A. Three-Level Architectures

One of the first integrated intelligent agent systems was a collection of computer programs and hardware known as "Shakey the Robot" (Nilsson, 1984). Shakey's design was an early example of what has come to be called a *three-level architecture*. The levels correspond to different paths from sensory signals to motor commands.

At the lowest level of such architectures are actions that use a short and fast path from sensory signals to effectors. Important "reflexes" are handled by this pathway—such as "stop" when touch sensors detect a close object ahead. Servo control of motors for achieving set-point targets for shaft angles and so on are also handled by these low-level mechanisms.

The intermediate level combines the low level actions into more complex behaviors—ones whose realization depends on the situation (as sensed and modeled) at the time of execution. This level uses more abstract (or more "coarse") perceptual predicates and more complex actions than do the lower ones. Whereas reflex actions are typically evoked by primitive sensory signals, the coordination of intermediate-level actions requires more elaborate perceptual processing.

The third level usually involves systems that can generate plans consisting of a sequence of intermediate level programs.

The three-level architecture has been used in a variety of robot systems. As a typical example, see (Connell, 1992).

B. The Triple-Tower Architecture

A generalization of the three-level architecture has been proposed by Albus and colleagues (Albus, 1991; Albus, McCain, Lumia, 1989). They envision hierarchies or "towers" of perceptual, modeling, and action processing. We propose here a particular instantiation of their triple-tower architecture.

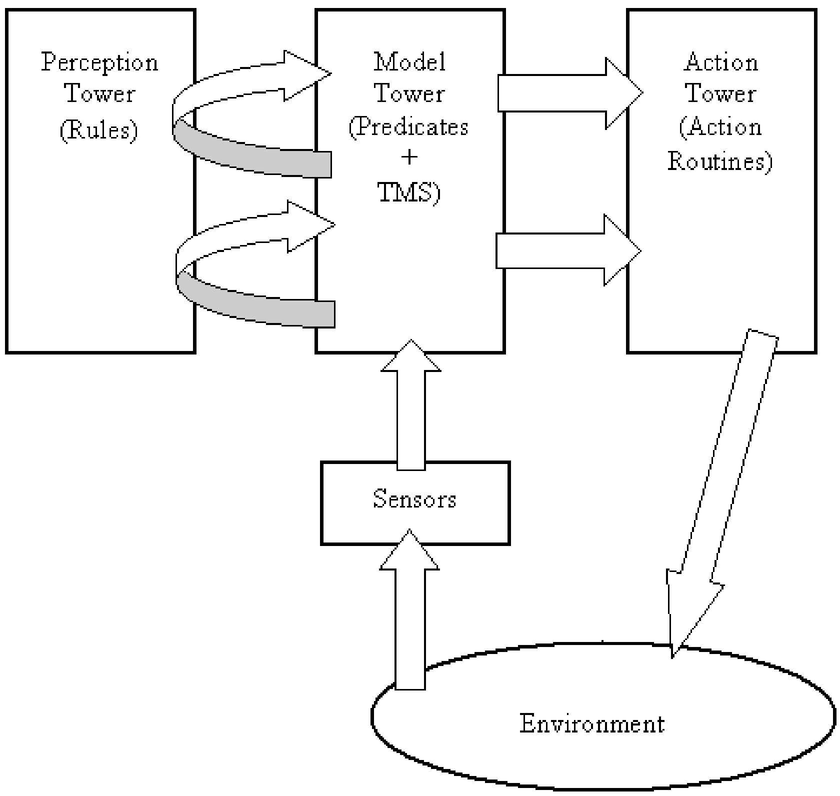

Figure 1: A Triple-Tower Architecture

The novel features of our proposal are:

- The use of teleo-reactive programs in the action tower
- The use of perceptual rules in the perception tower. These rules create increasingly abstract predicates from simpler ones
- The use of a truth-maintenance system (TMS) to keep the predicates in the model tower continuously faithful to changes in the sensed environment

My version of this triple-tower architecture is illustrated in Figure 1. The operation of such a system would proceed as follows: Aspects of the environment that are relevant to the agent's roles are sensed and converted to primitive predicates and values. These are stored at the lowest level of the model tower. Their presence there may immediately evoke primitive actions at the bottom of the action tower. These actions, in turn, affect the environment, and some of these effects may be sensed—creating a loop in which the environment itself might play an important computational role.

The perception tower consists of rules that convert predicates stored in the model tower into more abstract predicates which are then deposited at higher levels in the model tower. These processes can continue until even the highest levels of the model tower are populated.

The action tower consists of a loose hierarchy of action routines that are triggered by the contents of the model tower. The lowest level action routines are simple reflexes—evoked by predicates corresponding to primitive percepts. More complex actions are evoked by more abstract predicates appropriate for those actions. High-level actions "call" other actions until the process bottoms out at the primitive actions that actually affect the environment.

We also allow for the possibility that the actions themselves might affect the model tower directly (in addition to the loop through the environment) by writing additional and/or altered content. With the ability both to read from and write in memory, the triple-tower structure is a perfectly general computational architecture.

In order to be responsive to ongoing environmental changes, we include a truth-maintenance system (TMS) as part of the model tower. Such a system should continuously delete predicates and values from the model tower that are no longer derivable (through the perceptual rules) from the then-present components of the model tower. We describe a specific example of such a triple-tower system in section III.

II. Teleo-Reactive Programs

I propose that the actions be implemented as teleo-reactive (T-R) programs. For completeness, we give a brief overview of this formalism here.

A *teleo-reactive (T-R)* program is an agent control program that robustly directs the agent toward a goal in a manner that continuously takes into account changing perceptions of the environment. T-R programs were introduced in two papers by Nilsson (Nilsson 1992, Nilsson 1994). See also the T-R web page at:

`www.robotics.stanford.edu/users/nilsson/trweb/tr.html`

In its simplest form, a T-R program consists of an ordered list of production rules:

$K_1 \rightarrow a_1$

...

$K_i \rightarrow a_i$

...

$K_m \rightarrow a_m$

The K_i are conditions, which are evaluated with reference to a world model. The a_i are actions on the world (or that change the model). 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.

A T-R 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 T-R program is usually 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 will achieve some particular condition, say K_j, higher in the list (that is, with $j < i$). T-R programs designed in this way are said to have the *regression property.*

We assume that the set of conditions K_i (for $i = 1, ..., m-1$) covers *most* of the situations that might arise in the course of achieving the goal K_1. In any case, we can set $K_m = T$ (or 1) as a default catch-all. If an action fails to achieve its expected condition, due to an execution error, noise, or the interference of some outside agent, the program will nevertheless typically continue working toward the goal. This robustness of execution is one of the advantages of T-R programs.

T-R programs differ substantively from conventional production systems, however, in that actions in T-R programs can be durative rather than discrete. A *durative* action is one that can continue indefinitely. For example, a mobile robot might be capable of executing the durative action *move*, which propels the robot ahead (say at constant speed). Such an action contrasts with a discrete one, such as move forward one meter. In a T-R program, a durative action continues only so long as its corresponding condition remains the highest true condition in the list. When the highest true condition changes, the current executing action immediately 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 when its associated condition ceases to be the highest true condition.

The regression condition for T-R 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 a general T-R program, the conditions K_i may have free variables that are bound when the T-R 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. Actions in a T-R program may be primitive, they may be sets of actions executed simultaneously, or they may themselves be T-R programs. Thus, recursive T-R programs are possible. (See Nilsson 1992 for examples.)

When an action in a T-R program is itself a T-R program, it is important to emphasize that the usual computer science control structure does *not* apply. The conditions of *all* of the nested T-R programs in the hierarchy are *always* continuously being evaluated! The action associated with the highest true condition in the highest program in the stack of "called" programs is the one that is evoked. Thus, any program can always regain control from any of those that it causes to be called—essentially interrupting any durative action in progress. This responsiveness to the current perceived state of the environment is another one of the advantages of T-R programs.

Sometimes it is useful to represent a T-R program as a tree, called a T-R tree, as shown in Figure 2.

Suppose two rules in a T-R program are $K_i \rightarrow a_i$ and $K_j \rightarrow a_j$ vith $j < i$ and with K_i the regression of K_j through action a_i. Then we have nodes in the T-R 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, should 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

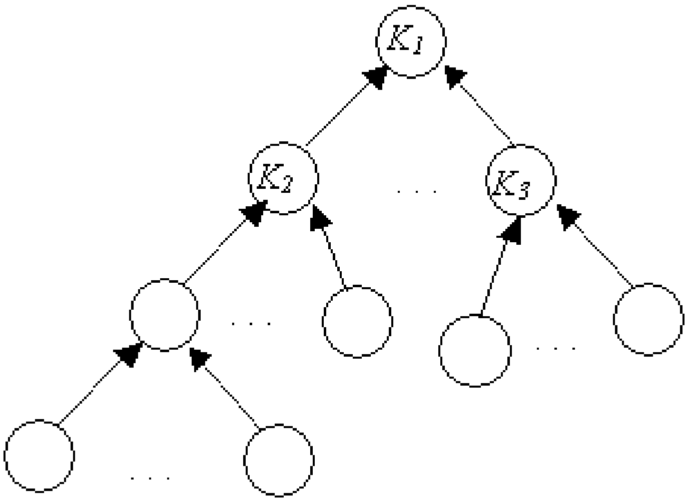

Figure 2: A T-R tree

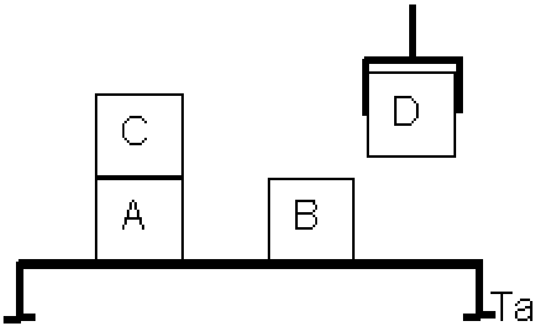

Figure 3: Blocks on a Table

actions for achieving the parent's condition.

Continuous execution of a T-R tree would be achieved by a continuous computation of the shallowest true node and execution of its corresponding action. (Ties among equally shallow true nodes can be broken by some arbitrary but fixed tie-breaking rule.)

The "backward-from-the-goal" approach to writing T-R programs makes them relatively easy to write and understand, as experience has shown.

III. A Triple-Tower T-R Agent Builds a Tower

As an illustrative example, I describe a triple-tower system for building a tower of blocks on a table. We assume the agent works in an environment of blocks and a table as shown in Figure 3.

In this example, there are four blocks (A, B, C, and D), a table (Ta), and a hand that can move blocks from place to place. In the sketch, the hand is holding block D.

At the lowest level of the action tower are the action schemas $putdown(x, y)$

and $pickup(x)$. The schema $putdown(x, y)$ can be applied if the hand is holding block x and if y is either the table or a block with no other block on it. The result (if successful) is that block x is on y (that is, either directly on the table or on top of block y). The schema $pickup(x)$ can be applied if no block is on block x and the hand is not already holding anything. The result is that the hand will be holding block x. We assume that these actions are "ballistic." They continue executing until they terminate (either successfully or in some type of failure).

The sensory system creates all of the instances of the predicates $On(x, y)$ and $Holding(x)$ that are satisfied by the environment. These are immediately placed at the bottom of the model tower. (These predicates have their obvious intended interpretations.) If applied to the environment of blocks and table shown above, the sensors would create and place in the model tower the following assertions: $On(C, A)$, $On(A, Ta)$, $On(B, Ta)$, and $Holding(D)$. These assertions remain in the model tower only so long as direct sensory perception of the environment sustains them. (For example, if block C is moved, the assertion $On(C, A)$ would be immediately deleted.)

The goal of our block-stacking agent is to make a tower of blocks with all of the blocks in the tower ordered from top to bottom as specified by a user. The user specifies the order, from top to bottom, by a list, such as (A, B, C). The complete system for this example is shown in Figure 4.

The perception tower is composed of a number of rules that are invoked in the forward direction to create new predicates—more abstract than the primitive ones, which are created directly by the sensors. These rules are invoked by predicates that are already in the model. The results are inserted in the model. For example, the rule

$$\neg(\exists x)On(x, y) \wedge \neg Holding(y) \supset Clear(y)$$

can be used to assert $Clear(C)$ in the model tower because (by reference to the model) there is no assertion of the form $On(x, C)$ and there is no assertion $Holding(C)$. (We assume "negation as failure.") Similarly, we can assert $Clear(B)$.

The rule

$$On(x, Ta) \rightarrow Ordered(list(x))$$

can be used to assert $Ordered((A))$ and $Ordered((B))$ in the model because both $On(A, Ta)$ and $On(B, Ta)$ are in the model already. (The Lisp construct $list(x)$ creates a list whose single element is the argument x. Note our use of other Lisp notation throughout the example.) Using the various rules in the perception tower, more abstract predicates come to populate the model tower. The model tower is used to evaluate predicates as needed by the T-R programs in the action tower. Keep in mind however, that all of these model assertions are subject to a truth-maintenance system that continuously updates them as the (sensed) environment changes.

A user can specify a tower to be built by calling the top-level T-R program, $maketower(x)$, with x instantiated to whatever tower s/he desires. For example, s/he might call $maketower((A, B, C))$. Calling this program invokes other T-R programs in the action tower. Their invocation ultimately results in the execution of primitive actions whose primitive effects are sensed and recorded in the model tower—leading to the insertion and retraction of other

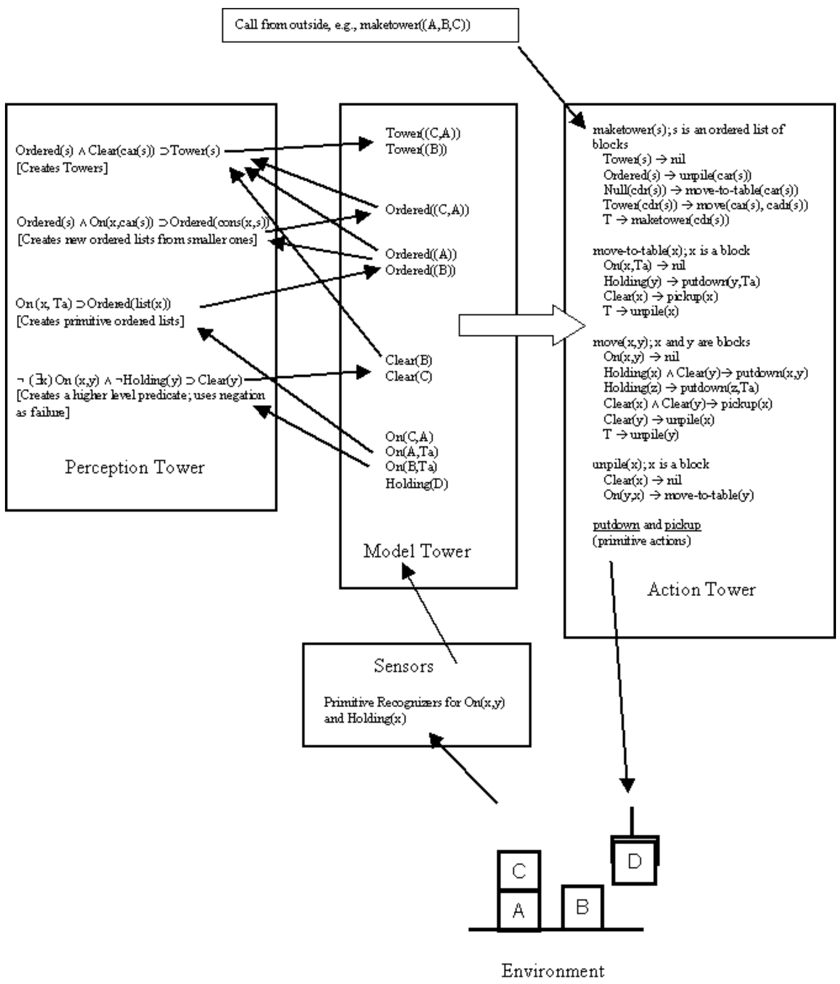

Figure 4: Triple-Tower Architecture for Building a Tower

predicates and the evocation of other actions. The reader is invited to trace through a successful execution. Note that the evocation of T-R programs is not strictly hierarchical. Lower level programs can call those at a higher level. Note also that the top-level program, maketower, is recursive. Remember that called programs are allowed to continue only so long as the relevant condition in the program that called them remains in the model tower—throughout the entire stack of T-R programs invoked at any time.)

The architecture just described is capable of building any single tower without the need for search. The complexity is polynomial in the size of the tower. [Compare with a similarly efficient scheme for tower building proposed by (Chapman, 1989). The "blocks world" is NP-hard only for building multiple towers (Gupta & Nau, 1992).]

A Java applet [2] implementing this triple-tower, T-R system is available on the web at:

`www.robotics.stanford.edu/`
`users/nilsson/trweb/TRTower/TRTower.html`

IV. The Vision

Our example shows that it is relatively straightforward to design a triple-tower T-R system for simple, demonstration-type tasks. The challenge is to see whether or not such systems can be designed for the many real-world tasks that we want robots and agents to perform. As we have seen, there are three major components to be synthesized, namely, abstractions of sensed data, rules for creating these abstractions, and hierarchies of T-R programs. The abstractions, rules, and programs are all interdependent, which can make synthesis of complex programs difficult.

In addition to the manual methods of synthesis (such as was employed to develop the block-stacking system), we can also employ automatic planning and learning techniques. By themselves, planning and learning methods are probably not sufficient for the synthesis of large and effective robot control programs *ex nihilo.* I believe that efforts by human programmers at various stages of the process will continue to be important—initially to produce a preliminary program and later to improve or correct programs already modified by automatic planning and learning techniques. I envision that four methods, namely programming, teaching, reinforcement learning, and planning might be interspersed in arbitrary orders. It will be important therefore for the language(s) in which programs are constructed and modified to be languages in which programs are easy for humans to write and understand and ones that are compatible with machine learning and planning methods. These requirements most likely rule out, for example, C code (it would be hard to learn) and neural networks (they would be hard for humans to understand and modify), however useful these formalisms might be in other applications. T-R programs, on the other hand, are easy for humans to write and modify, and there has already been some progress on their synthesis by automatic planning and learning methods.

Benson's TRAIL system produces T-R programs via a planning system that employs STRIPS-type rules learned by inductive logic programming (IPL) methods (Benson 1996). As regards other varieties of learning, we might distinguish two types, namely teaching and reinforcement learning. *Teaching* involves someone showing the robot what is required by "driving" it through various tasks. This experience produces a training set of conditions and corresponding actions, which can then be used by supervised learning methods to clone the behavior of the teacher in the form of T-R programs. *Reinforcement learning* involves on-the-job trials guided by rewards given by a human user or teacher, and/or by the environment itself. Preliminary work on the use of teaching and reinforcement learning to produce T-R programs is described in (Nilsson 2000).

There is also other research that might be relevant to the hierarchical aspects of learning triple-tower systems. Pfleger has investigated on-line techniques for detecting frequently occurring, hierarchically related sub-strings within data streams (Pfleger, 2001). Drescher has proposed methods for building concepts based on Piagetian cognitive development (Drescher, 1991). Stone has developed a layered learning architecture for robotic soccer players (Stone, 2000). And Dietterich proposes methods for hierarchical reinforcement learning (Dietterich 1998).

I think the triple-tower architecture provides an interesting setting for an attack on the problem of synthesizing mid-level control systems for robots and other agents.

References

Albus 1991: Albus, J. S., "Outline for a Theory of Intelligence," *IEEE Systems, Man, and Cybernetics*, 21(3):473-509, May/June 1991.

Albus, McCain, Lumia 1989: Albus, J. S., McCain, H. G., and Lumia, R., "NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM), NIST Technical Note 1235, 1989 Edition, National Institute of Standards and Technology, Gaithersburg, MD, April 1989 (supersedes NBS Technical Note 1235, July 1987).

Benson 1996: Benson, S., *Learning Action Models for Reactive Autonomous Agents*, PhD Thesis, Department of Computer Science, Stanford University, 1996. (Postscript version available at:
http://robotics.stanford.edu/users/sbenson/thesis.ps.)

Chapman 1989: Chapman, D., "Penguins Can Make Cake," *AI Magazine*, 10(4):45-50, 1989.

Connell 1992: Connell, J., "SSS: A Hybrid Architecture Applied to Robot Navigation," in *Proc 1992 IEEE International Conf. on Robotics and Automation*, pp. 2719-2724, 1992.

Dietterich 1998: Dietterich, T. G., "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition," in *Proceedings of the 15th International Conference on Machine Learning ICML'98*, San Francisco, CA: Morgan Kaufmann. (Also to appear in the *Journal of Artificial Intelligence Research.*)

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

Gupta & Nau 1992: Gupta, N., and Nau, D., "On the Complexity of Blocks-World Planning," *Artificial Intelligence*, 56(2/3):223-254, 1992.

Nilsson 1984: Nilsson, N. J., *Shakey the Robot*, Technical Note 325, SRI International, Menlo Park, CA, 1984.

Nilsson 1992: Nilsson, N. J., *Toward Agent Programs with Circuit Semantics*, Technical Report STAN-CS-92-1412, Stanford University Computer Science Department, 1992.

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

Nilsson 2000: Nilsson, N. J., "Learning Strategies for Mid-Level Robot Control: Some Preliminary Considerations and Experiments," draft memo, May 11, 2000, Stanford University. (MS WORD version available at `www.robotics.stanford.edu/users/nilsson/trweb/tr.html` .)

Pfleger 2001: Pfleger, K., *Data-driven, Bottom-up Chunking: Learning Hierarchical Compositional Structure*, PhD Dissertation (in progress), Department of Computer Science, Stanford University, Stanford, CA 94305, 2001.

Stone 2000: Stone, P., *Layered Learning in Multi-Agent Systems: A Winning Approach to Robotic Soccer*, Cambridge, MA: MIT Press, 2000.

Footnotes

[1] Parts of this section are adapted from Chapter 25 of my book, *Artificial Intelligence: A New Synthesis*, San Francisco: Morgan Kaufmann, 1998.

[2] Running the applet requires the Java 2 Runtime Environment (JRE), version 1.3 or above. Visiting the applet's website should result in your being asked if you want this JRE downloaded to your computer. You can check the version number of your JRE by typing under the shell prompt: java-version. JRE v1.3 can be downloaded from Sun; it is available for the platforms: Microsoft Windows, Linux (x86), and Solaris (SPARC/x86).