TRIANGLE TABLES: A PROPOSAL FOR A ROBOT PROGRAMMING LANGUAGE

Technical Note 347

February 1985

By: Nils J. Nilsson, Senior Staff Scientist

Artificial Intelligence Center
Computer Science and Technology Division
SRI International

Present (1985) affiliation: Chairman, Computer Science Department,
Stanford University, Stanford, California 94305

I Introduction

Structures called *triangle tables* were used in connection with the SRI robot *Shakey* [1] [2] for storing sequences of robot actions. Because the rationale for triangle tables still seems relevant, I have recently elaborated the original concept and have begun to consider how the expanded formalism could be used as a general robot-programming language. The present article describes this new view of triangle tables and how they might be used in a language that supports asynchronous and concurrent action computations.

Triangle tables can be thought of as a specialized way of writing production rules. A robot system based on production rules might have components as shown in Figure 1.

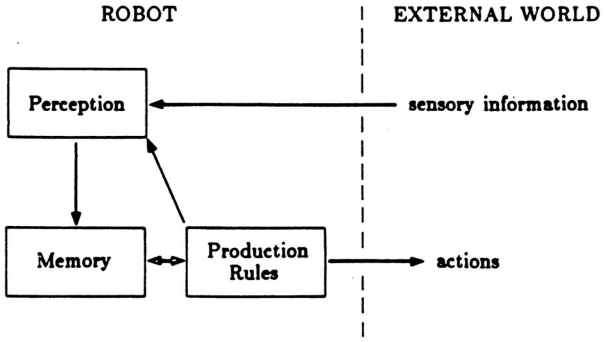


Figure 1 Components of a Simple Robot System

In our conception of a robot system, the production rules operate by testing a condition on the robot's memory. If the condition is satisfied, the action part of the rule is executed. The action may have an effect on the external world through robot effectors and/or it may alter the memory or the perception system. (It is even possible that the action may change the production rules themselves, although we do not consider such a possibility in this note.) We imagine that the memory is a collection of first-order formulas, typically ground literals. The condition part of the production rule is a logical formula, and the memory test consists of determining whether or not this formula follows from the literals in memory.

We imagine that sensory information affects robot actions by virtue of being recorded as formulas in memory. The perception system converts sensory signals into these formulas. Thus, the production rules do not test the world "directly," but rather are sensitive only to a representation or "model" of the world in memory.

In a special case of such a system, the production rule conditions are all mutually exclusive and exhaustive. That is, the memory and rules are organized so that the memory always satisfies *precisely one* of the production rules. The system then is readily seen to be a finite-state machine whose states correspond to memory equivalence classes defined by each of the production rule conditions. We shall not limit ourselves to this special case, however.

Production rule formalisms have several advantages for robot systems. First, they are simple and modular. Changes can be made by adding, deleting or modifying rules without having to change the production rule interpreter. Second, they can have short "sense-act" cycles and are therefore not as blind to changes in the world as are systems committed to lengthy actions that cannot be modified by sensory data. Thus, production rule systems are robust in that undesired effects of an inappropriate or abortive action can be noticed quickly so that corrective action can be initiated. Third, they invite parallel implementations. In principle, each rule could be implemented by dedicated hardware and the entire ensemble of rules could be performing memory tests and executing actions asynchronously and simultaneously.

There are two problems with production systems, however, that have limited their application. First, their sensory-driven nature seems to militate against "goal-directed" behavior. Second, there have been difficulties in developing hierarchical organizations of production systems. Hierarchical design is especially important for complex systems; most production systems seem to be too "flat." In designing hierarchical production systems, one has to be careful not to sacrifice the short sense-act cycle. (If the "action" of a production rule is to "call" another production system, the calling system surrenders control until it is explicitly returned by the called system.) Triangle tables maintain all the advantages of production systems while allowing goal-directed behavior and hierarchical organization.

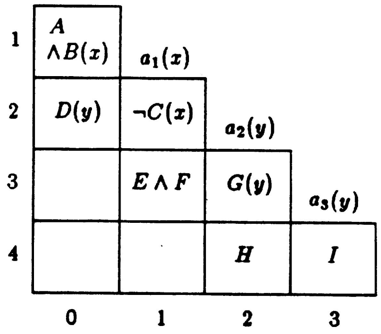
II Syntax

Programs consisting of robot actions are designed to achieve certain sequences of effects—some of which are related to the purposes of the programs, while some merely enable subsequent actions to be executed. Information about the preconditions and effects of each robot action within a larger program and how these actions are interrelated can be conveniently represented in a triangular table. Interestingly, this very table can be used as a representation of the program itself. The syntax and semantics of these tables will make clear what they are and how they are used.

A triangle table, of *rank* N, is an $N \times N$ triangular array of cells; each cell may contain a collection of formulas, which latter might contain schema variables. The rows of the array are numbered from the top, starting with row 1; the columns of the array are numbered from the left starting with column 0. Each column except the $0th$ is headed by an action schema. The schema variables in any action schema are a subset of those formula schema variables present in the cells in the row to the left of that action schema. The schema variables in any formula are a subset of those action schema variables present in the action (if any) heading the column in which that formula appears.

We say that the conjunction of the formulas in the last row of the table is the *effect* of the table and that the conjunction of formulas in the $0th$ column of the table is its *precondition.*

We illustrate a typical triangle table in Figure 2.



1	A $\wedge B(x)$	$a_1(x)$		
2	$D(y)$	$\neg C(x)$	$a_2(y)$	
3		$E \wedge F$	$G(y)$	$a_3(y)$
4			H	I
	0	1	2	3

Figure 2 A Typical Triangle Table

The expressions $A \wedge B(x), D(y), \neg C(x), E \wedge F, G(y), H, I$ are formula schemas; the expressions $a_1(x), a_2(y), a_3(y)$ are action schemas.

III Semantics

Before discussing how a triangle table is used as a program of robot actions, we first explain what the formulas in a table are intended to mean and why they are placed as they are. The conjunction of all the formulas in the row to the left of an action is a precondition for executing that action. More precisely, if an instance of this conjunction follows from the formulas in memory, the corresponding instance of the action can be executed. The conjunction of the formulas in the column immediately beneath an action is the presumed effect (on memory) of that action. More precisely, after an instance of that action is executed, then the corresponding instance of the formulas in that column can be presumed to follow from memory. The only formulas underneath an action in a triangle table are those effects that are also either preconditions of actions heading higher-numbered columns or effects in the last row of the table. The formulas representing the effects of actions are distributed among the column cells in such a way that those that are preconditions of other actions are in a cell to the left of that action. It is possible that a formula may be repeated in a column if it is part of the precondition of more than one action.

In Figure 2, for example, $E \wedge F \wedge G(y)$ is a precondition for $a_3(y)$ in the sense that, if an instance of the precondition is true, then the corresponding instance of a_3 can be executed. Instances of the formula $\neg C(x)$ are effects of corresponding instances of the action $a_1(x)$; $E \wedge F$ is an additional effect of all instances of $a_1(x)$. H and I are effects of a_2 and a_3, respectively, which are not preconditions of any actions named in the table.

It is intended that a triangle table program will be *typically* executed by executing actions in the sequence $\{a_1, a_2, \ldots, \}$, where a_i is the action heading the ith column. (We say *typically* because, as we shall see, one of the features of triangle tables is that we can deviate from this sequence when appropriate.) The intention is that preconditions of subsequent actions might be among the effects achieved by a_i. Those effects that are not accomplished solely to achieve preconditions of subsequent actions are represented by formulas in the last row of the table. These are the final effects of the table. Preconditions of actions that are not realized by any actions in the table must hold before the table is executed; these preconditions are represented by formulas in the $0th$ column of the table. These are the initial preconditions of the table. To be listed in the table, any effect $\mathcal{F}$ needed by action a_j and provided by a previous action a_i must survive the intervening actions. An effect represented by a formula in the last row of the table survives all actions subsequent to the one that achieved it.

The conjunction of the formula schemas in the rectangular subarray consisting of the bottom $N-(n-1)$ rows of the leftmost n columns is called the *nth kernel* of the table. The second kernel of the table in Figure 2 is $D(y) \wedge \neg C(x) \wedge E \wedge F$. The third kernel is $E \wedge F \wedge G(y) \wedge H$. Each kernel can be thought of as the precondition that must hold for a certain sequence of actions to be executable and for the effects of the table to be achieved. Instances of the ith kernel are preconditions for corresponding instances of the action schema sequence $\{a_i, \ldots, a_N\}$ to be executable and to achieve the effects that appear in the last row of the table. If an instance of the Nth kernel follows from memory, then without executing any actions, a corresponding instance of the table's effects from memory also follows. The first kernel, which is the precondition of the entire table, must have a true instance for the corresponding instance of the entire sequence of actions to be executable and to achieve the table's effects.

IV Executing Triangle Tables

For the moment, we will think of triangle tables as programs in the form of data structures that are to be executed by an interpreter. (Later we shall discuss the possibility of embedding these programs and their interpreter(s) in parallel-operating hardware.)

The operation of the interpreter involves checking the various kernels of a triangle table. Instances of kernels may or may not follow from the formulas in memory. The highest-numbered kernel in the table having an instance that does follow from memory (if there is such a kernel) is called the *active* kernel. Action sequences associated with active kernel instances are called *active* sequences. The first action in an active sequence is called an *active* action. Whenever there is an active kernel, executing a corresponding active sequence will achieve the corresponding instance of the effects of the table.

Since actions do not always achieve their intended effects, and since the world may change independently of what the robot does (albeit with perceived changes recorded automatically in memory), we would like to make sure that robot actions are chosen with a close eye on memory. Thus, when we desire to achieve the effects of a triangle table, we would rather not commit ourselves to executing an entire active sequence of that table. For if we did, unanticipated, sensory-induced memory changes partway through the sequence might render the remaining actions inappropriate (perhaps unnecessary or even harmful). We would rather just execute an active action, wait to allow the perceived world to have its effects on memory, and then recompute an active action to be executed and so on. This subsequent active action will often be the same as the second action in the active sequence computed earlier. In these cases, recomputing an active action might be thought of as unnecessary computational effort, but the need to react appropriately to the uncertainties of many robot environments justifies the extra computation. In any case, if the tables are kept relatively small, the extra computational effort can be controlled.)

Thus, in summary, to execute a triangle table means to compute and then execute an active action. If the active kernel is the last kernel, execution of the table ceases.

A somewhat inefficient way to compute the active kernel is first to determine whether the Nth kernel has an instance that is true. If it does not, then verifywhether or not the $(N-1)$-*st* kernel has an instance that is true, and so on. The reason this procedure is inefficient is that the kernels have several cells of the table in common, and the formulas in these common cells might have to be checked several times. There is a more efficient procedure that checks each cell only once. We explain how it works for the special case in which the triangle table formulas have no schema variables. We begin with cell $(N, 0)$ and scan along the bottom row until we find a cell whose formula is *not* true. Placing a boundary just to the left of this cell, we scan the next-to-the-bottom row, starting with cell $(N-1, 0)$. If we reach the boundary before finding a cell whose formula is not true, we begin scanning the next-higher row; if we find a cell with a formula that is not true, we move the boundary to its left and then proceed to scan the next-higher row, and so on. As soon as the set of scanned cells comprises all the cells of a kernel, we have found the active kernel.

To illustrate how a triangle table program works, we consider a simple, if fanciful, example. Consider a robot with the following primitive actions:

goto(x)—the robot goes to place x

pickup(x)—the robot picks up object x

wait—the robot waits and does nothing

hand(y, x)—the robot hands object x to person y

With these actions, the intended effect of the triangle table in Figure 3 is $HAVE(y, x)$—person y has object x. In other words, this is a triangle table for delivering objects to people, so let's call it **deliver(y,x)**.

	0	1	2	3	4	5
1		goto $(p(x))$				
2		$AT(R,p(x))$	pickup(x)			
3				goto $(o(y))$		
4	$H(y)$			$AT(R, o(y))$	wait	
5			HAVE(R, x)		$AT(R,p(y))$	hand(y, x)
6						HAVE (y, x)

Figure 3 The Triangle Table deliver(y,x)

The predicates and functions of Figure 3 have the following intended meanings:
$AT(R, x)$—The robot R is at place x
$H(y)$—Person y is at work today
$HAVE(y, x)$—Agent y has object x
$p(x)$—The present location of object or person x
$o(y)$—The usual location (say, the office) of person y
R—The robot

We assume that the memory has information in it that allows the robot to determine where it, objects, and people are located. (Or at least it has information that allows it to establish the truth of the predicates in the table when they are in fact true.) The intended sequence of actions prescribed by this table is for the robot to go to the object's location, pick the object up, go to the office where the recipient usually is, wait for the recipient (if necessary), and then hand over the object. The only precondition for this table is that the intended recipient be at work today.

A triangle table is *called* when it is desired to achieve an instance of its effect. Suppose we want a robot to achieve the condition $HAVE(JOHN, PAYCHECK)$. We can do this by calling the instance **deliver**$(JOHN, PAYCHECK)$.

Even this rather simple example illustrates two important features of this style of programming. One is *opportunism.* If, for example, the memory ever indicates that $HAVE(JOHN, PAYCHECK)$ is true (for whatever reason), the last kernel will be the active one. Thus, if, when the robot goes to pick up John's paycheck, it is informed that John already has it, the table will come to an appropriate halt.

Similarly, if conditions in the table are satisfied independently of the robot's own actions, the robot will take advantage of such good fortune and not execute any superfluous actions.

Another feature is the robustness of triangle tables. If an action does not achieve its intended effect, but instead results in some other effect, the robot will not blindly carry out the next action in the table. Instead the interpreter may call for re-execution of the unsuccessful action or, perhaps, the execution of some instance of another action. Additional features will become apparent in the next section.

V Hierarchical Systems of Triangle Tables

So far we have treated the actions that head the columns in triangle tables as primitive. But these actions themselves could have internal structure; indeed, they could be programmed by using triangle tables. When nonprimitive actions are executed, we have control options that are more complex than the one associated with merely executing a primitive action. As would be the case with primitive actions, we could *transfer control* completely to the program that executes the action—and not regain control until it is explicitly returned. If the action is programmed as a triangle table, there are interesting ways to transfer control in a much more tentative fashion. For example, we could transfer control to the *called* triangle table to allow just the active action in the subordinate table to be executed. Doing this might cause changes in memory that result in a new active action in the *calling* table. We could follow this scheme all the way down through a hierarchy of tables until primitive actions are reached. In such a hierarchy, the "top" table always has control; action selection passes down through the hierarchy until a primitive action is selected and executed, whereupon the top table starts the process over again.

The advantage of this sort of control regime is that active actions are computed for all of the calling tables in the hierarchy after each primitive action is executed. Thus, the actions are always *relevant* to the current state of the memory—the feedback loops are all short. If conditions in the world change in unanticipated ways (provided that these changes are perceived and recorded in memory), the system can react to them appropriately; it never commits itself to "open-loop" execution of action sequences.

One interesting way to achieve such a control regime is to have all triangle tables run asynchronously and concurrently. Let us suppose that we have parallel hardware to implement each table and its interpreter. The interpreter for a table is always computing the active kernel and an instance of an active action. The interpreter is essentially declaring something like $a(\sigma)$ *is now an active action instance*, where a is the name of the action and σ is a collection of schema variable instances. That fact, like everything else, can be perceived and recorded in memory. Once there, it is a declarative statement in logic represented by the formula, $AA(a, \sigma)$. (Just like other statements inserted into memory by the perception apparatus, we assume that these statements are kept "current." That is, when $a(\sigma)$ ceases to be an active action instance, $AA(a, \sigma)$ is automatically erased from memory.)

To complete our adaptation of triangle tables to hierarchical and parallel operation, we next insert into the bottom left-hand cell of each table a condition of the form $AA(u, v)$, where u is the name of the action programmed by the table and v represents the schema variables associated with that action. (For simplicity, the notation of this explanation is the one that is appropriate when there is just a single schema variable. When there is more than one schema variable, we give AA more arguments. That is, if v_1 and v_2 are schema variables, we write $AA(u, v_1, v_2)$.) With the condition $AA(a, v)$ in the bottom left-hand cell of the table that implements action $a(v)$, the insertion of the formula $AA(a, \sigma)$ into memory makes it possible for one of the kernels of the $a(v)$ table to be active. That is, putting $AA(a, \sigma)$ into memory acts like a *call* to execute $a(\sigma)$. Without $AA(a, \sigma)$ in memory, the $a(v)$ table will remain inactive—even though its interpreter is attempting unsuccessfully to find an active kernel. For example, the table of Figure 3 would have in its bottom left-hand cell the formula $AA(deliver, y, x)$. No kernels of the **deliver** triangle table could then be active without an instance of $AA(deliver, y, x)$ present in memory. In the absence of such an instance, there is no active instance of **deliver** and therefore no valid reason to execute the table.

A collection of triangle tables would be linked by those active action instances that are associated with each of their kernels. Their interpreters would compute active action instances asynchronously and concurrently—with the proper formulas being inserted into memory. These computations would bottom out in the selection of primitive actions, which would then be executed. As long as the primitive actions do not conflict, they too can be executed asynchronously and concurrently. (These actions would bring about changes in the world, which would be perceived and affect memory.) Provision would have to be made for resolving any conflicts among primitive actions.

The flexibility and robustness of the triangle table idea is particularly apparent in this scheme. Assume, for example, that, in running the **deliver** triangle table of Figure 3 (with arguments y and x bound to *JOHN* and *PAYCHECK* respectively, the robot meets (and perceives) John on its way to his office. That is, during execution of the triangle table **goto**$(OFFICE, JOHN)$, there appears in memory the expression $AT(R, p(JOHN))$ (or formulas that allow derivation of that expression). Execution of **goto**$(OFFICE, JOHN)$ would cease abruptly because the kernel supporting it would no longer be active. Instead, the kernel supporting **hand**$(JOHN, PAYCHECK)$ would become active.

To initiate activity among a collection of triangle tables, an AA formula must somehow be inserted into memory. This can be done either by some external agent (such as a human user of the robot system) or by some *top table* that does not have an AA condition and can become active by other means. If a human user wants the robot system to execute an action implemented by one of the tables, he or she has merely to insert the corresponding AA formula into memory. If assembled in a hierarchical and noncyclic fashion, a collection of triangle tables would continue to execute primitive actions until there were no active kernels left. This halting condition might result, for example, when the top table has only its highest-numbered kernel active, i.e., the one that has no action associated with it. The entire collection of tables, interpreters, perception, and memory can then be regarded as a complex set of feedback circuits operating relentlessly to satisfy the bottom row of the top table.

VI Future Work

I have not yet worked out all the diverse mechanisms by which the contents of memory might be changed. We have discussed the fact that memory is affected by the perceptual apparatus of the robot. It is also affected by the addition (and deletion) of AA formulas by the triangle table interpreters. It might prove useful to have the primitive actions themselves insert certain formulas into memory. (Some of these primitive actions might have such reliable effects that we could assume them without benefit of perception.) Moreover, the triangle table interpreters could perhaps affect memory in ways other than just adding AA formulas.

We have assumed that each kernel in a triangle table has associated with it a single action, which might be primitive or might be programmed as a triangle table. We could relax this assumption by allowing a certain amount of nondeterminism in action selection. That is, we might associate two or more actions with a single kernel. A form of nondeterministic action selection might then be accomplished by a scheme that involved adding to memory a conjunction of AA formulas (with each conjunct corresponding to one of the actions) and sorting out any conflicts at the primitive-action level.

There is also interesting work to be done on the stability properties of triangle table programs. Under what conditions will they halt? Under what conditions will they cycle? Some repeated action sequences will be useful and may be required to overcome overly stubborn or resilient problems. Repetition will also be necessary when primitive actions, although usually achieving their intended effects, do so unreliably. In any case, a useful system will have to cease repetitive activity eventually and special mechanisms may be employed to break cycles.

Triangle tables might also be useful as a general computational formalism. For computations that create and manipulate data structures, we could imagine a fictitious "world" of data structures (not to be confused with "memory") that are affected by the primitive actions. Through "perceptual" operations on the data structures, the memory would model whatever features of this world were important. The reader can easily create some simple triangle table programs to do list structure manipulations. The fact that triangle tables tend to repeat an action until a certain condition is met can be exploited to implement iteration. A program that is recursive in a function f can be achieved by associating with a kernel of the table implementing f a program that is an instance of f. Branching mechanisms are inherent in active-kernel computations. Triangle table programs may have advantages in applications other than robotics. We might even want to program some of the robot system's internal computations, such as perception and active-kernel computation, in the triangle table language. Perhaps the potentially extravagant use of asynchronous triangle table processors could be limited by controlling necessary parallelism through schemes that assign tables to processors.

(Georgeff [3] has speculated about how *correctness assertions* in ordinary programs might be used in a way similar to our use of kernels. Given this viewpoint, triangle table programs would be a special case of his model. Rosenschein and Pereira [4] are developing a new finite-state-automaton model of intelligent agents that shares some features with our triangle table model.)

Triangle tables were originally used as structures for storing sequences of robot actions computed by a planning system. We had in mind that they would play an important role in *learning* action sequences. This possibility should be investigated more thoroughly; it would seem advantageous to have a robot's fundamental actions programmed in the same language in which its more complex actions are learned.

Fikes, Hart and Nilsson [5] suggested a technique for splitting a triangle table into component tables that could operate concurrently and independently. They also hinted that the component tables could have more relaxed conditions for action initiation since each could depend on the other for fulfilling needed preconditions. I have not yet investigated how these proposals could be integrated with the hierarchical scheme proposed in this paper.

Production systems of various types have been proposed as models of human information processing. It would be enlightening to explore the extent to which the special features of triangle tables might make them useful as a basis for psychological models. They would seem particularly well suited for modeling animal activity in which purposeful and stimulus-driven behavior are combined. (There seems to be an intriguing connection between inserting *AA* formulas into memory and the *releasing* mechanisms of ethology. Readers interested in this subject might look at the models proposed by Deutsch [6] to see whether they could be made more concrete by using triangle tables.)

VII Acknowledgments

The late John Munson originally suggested grouping the conditions and effects of robot actions in a triangular table. During the late 1960s, John, Richard Fikes, Peter Hart, and I developed the triangle table formalism to represent sequences of robot actions. Several years elapsed before I began to think about applying an extended version of the formalism as a concurrent programming language for robot actions. I want to thank Michael Georgeff, Stan Rosenschein, and Leslie Pack Kaelbling for useful conversations during the writing of this article. The research reported here was partially supported by FMC under Contract 147466 (SRI Project 7390).

VIII REFERENCES

[1] Nilsson, Nils J., "Shakey the Robot," Technical Note No. 323, Artificial Intelligence Center, SRI International, Menlo Park, CA (April 1984).

[2] Fikes, Richard, Peter Hart, Nils J. Nilsson, "Learning and Executing Generalized Robot Plans," *AI Journal*, **3**(4), 251-288 (1972).

[3] Georgeff, Michael P. (private communication).

[4] Rosenschein, Stanley J. and Fernando C.N. Pereira, "Knowledge and Action in Situated Automata" (forthcoming).

[5] Fikes, Richard, Peter Hart, Nils J. Nilsson, "New Directions in Robot Problem Solving," *Machine Intelligence 7*, Meltzer, B. and D. Michie (eds.), Edinburgh University Press, Edinburgh, 405-430 (1972).

[6] Deutsch, J.A., "The Structural Basis of Behavior," The University of Chicago Press, Chicago (1960).