The State of State Tables in the AMRF

Don Libes

Integrated Systems Group
National Bureau of Standards
Gaithersburg, MD 20899

AMRF Technical Report #1
NBSIR 87-3541

ABSTRACT

This paper discusses the representation of automation control algorithms by state tables. Earlier papers favoring state tables are discussed in light of experience since they were written. It is suggested that the disadvantages of state tables outweigh their advantages and that we ought to look to more powerful representations.

This paper is concerned with representational adequacy for humans, and unless explicitly stated otherwise, the word "representation" will refer to this. This paper does not discuss "representations for computers".

Keywords: Automation Control Algorithms, State Tables, Algorithm Representation, Programming Languages

Background - The Automated Mauufacturing Research Facility

The National Bureau of Standards Automated Manufacturing Research Facility (AMRF) [Nanzetta84] is addressing issues related to computer/machine/robot/tool interface standards as they relate to small batch manufacturing. An automated factory is envisioned using a generic control schema at all levels. The factory at all levels is seen as a hierarchy much like a real factory or other natural hierarchical systems (such as a bureaucracy). Each process may communicate or control several processes but is controlled, itself, by only one other process.

The hierarchical structure of a factory can be represented as a tree with the facility controller at the root. Below the facility controller are shop controllers, which in turn command cell controllers. Below each cell controller are workstation controllers, and below this, equipment controllers. Each of these controllers is internally hierarchical. For example, the robot hierarchy includes controllers for an arm and gripper at a high level, while at the lower levels these are broken down into many individual controllers, one for each motor. For more information, see [Simpson82].

State Tables for Process Control

It is believed that computation can be broken down in this manner so that the levels themselves are computationally simple. Furthermore, each level can be in one of a very few states. [Barbera82] suggests 7±2, the size of human short-term memory according to many psychologists. One would be able to understand the states of such a state table in its entirety.

In practice, however, state tables are much larger than 7±2. We have programmed state tables requiring over a hundred states and several times as many transitions to achieve those states.

It is not clear if state tables are a relief from complexity at all considering [Barbera82], "...the entire input state, not just some subset of the input state, must be evaluated for each output. In order to implement a system that accomplishes this while remaining comprehensible, the system has been structured so that each control level is a state-table process where all of the inputs are sampled each time an output is to be generated."

[Barbera82] goes on to say that "The input state is defined by the data that encode the input command, the processed sensory information, the status from lower levels and the internal state values." These input values are matched against a preprogrammed set of values that select sets of outputs (via procedures) to be triggered.

In the AMRF, the number of variables referenced at one level of the hierarchy is large (say, 25 to 100). The researcher is thus potentially faced with building an immense state table. Needless to say, this is not what is done. In actual practice, several tricks are used to cut down on the number of variables appearing in the tables. Before a state table is evaluated, raw variables may be "preprocessed" into a much smaller number of state variables. This preprocessing (and a similar phase after state table evaluation called "postprocessing") step is programmed in a conventional high-level language. State selection during state table evaluation may also trigger execution of conventionally-programmed procedures.

The ability to execute conventional procedures is useful (and necessary). It can remove nitty-gritty details from the state tables and allow them to remain unencumbered from trivia. However, it is not clear what variables belong in the state table; or what decisions should be performed with the state table vs the pre- and post-processing steps. In general, researchers are inclined to push processing details into the conventional programming language when the the number of variables is unwieldy (say, greater than 7). Similarly, levels which are rather trivial encourage state tables to be filled with minutiae, since otherwise the tables would be virtually empty.

In other respects, a process' internal exposition is not unusual. Subroutines may be called by subroutines much as in any programming system. Only the highest routine must (and can) be a state table. Of what benefit is this?

Thus, state tables may always seem to be "about the right size" when in fact this goal has been artificially (and subconsciously) achieved. In empirical observation, the number of transitions (state table lines) tends to be of the order of the square of the number of input and state variables. For example, a state table with 6 variables tends to have close to 36 transitions.

$$\text{transitions} = O(\text{input variables}^2)$$

Figure 1 - Empirical observation of size of state tables

One way to reduce the large number of states that this implies, is to group related states into smaller tables which are selected based on a primary test of one variable. This was actually done in several of the projects at the AMRF that used state tables. Of course, the remaining tables could be split similarly (and split ...) until the system would look like it was programmed in a conventional programming language. I will discuss why or why not this is good.

State Tables - Advantages & Disadvantages

An early paper [Albus82] in the project cited many advantages for using state tables, that will be presented further on. Experience has shown that some of these advantages, while theoretically sound, do not work well in practice. Some claimed advantages are simply wrong. I will discuss and explain these as well as the real advantages of state tables, as I see them.

In [Albus82], state graphs, petri nets and state (transition) tables were all suggested as possible representations. The authors suggested algorithms could be represented as state graphs and by using a procedure (given in the paper) automatically be converted to state tables. However, the meat of the paper is spent citing advantages of using state tables themselves as appropriate representations for control algorithms. Seven advantages of state tables over conventional high-level languages (HLL) are elaborated. One disadvantage is mentioned.

I will discuss these advantages and disadvantages of state tables in the light of experiences using them. The claimed advantages of state tables are as follows:

> 1) "clarify and simplify design and synchronization of simultaneous processes."

State tables have no facilities for handling synchronization (cf. Petri nets). This must be handled by hand. For example, initiating n independent processes and then waiting for their completion requires $1+2^n$ states. (1 state to initiate them and 2^n to wait for them, since they may complete in any of 2^n orders.)

> 2) "at each cycle the system deals only with the present state and the present input. This enormously simplifies ... control ... because it partitions the problem domain and restricts it to a small number of variables with a limited time horizon."

There are several misleading ideas here.

First, the phrase "present state" means a subset of the entire present state. After all, with complete knowledge of the present (and enough computing power) you could accurately predict the future as well as have complete history from then on. The idea is to select a small number of variables, such that you are not computationally overwhelmed and yet can make informed decisions about what to do next. This idea is no different from high-level language programming. The assumption that programs have more context (history) than state-tables is a superficial view, probably based on the observation that the context of programs is more obvious than state tables.

Second, while decisions at any one moment rest upon a small set of variables, (a place for) each state table variable is present in every line of the state table whether it is germane or not. This may simplify control, but readability suffers.

Last, a limited time horizon is easier to achieve with state tables (as opposed to conventional programming languages) but this has its pitfalls, such as the *horizon effect* [Winston]. (See (3) for an example of this.)

> 3) "unanticipated error conditions ... can simply be ... added to the state table."

While it is possible to easily add lines in a state table for unanticipated states, such a "no match" condition may be indicative of a more serious fault, where the actual error occurred far back in time, but wasn't caught at that time. For example, indexing beyond the end of memory is not an indication that more memory is needed (the local view). Rather, it points to an error in a loop termination condition (the global view). (This is an example of the horizon effect, referred to in (2).) In practice, symptoms such as unanticipated states are not as easily fixed as by adding a line to a state table.

> 4) "new sensors or tests are easily handled by inserting a new line in the state table. ...inserting new lines into the tables ... does not alter the operation of the previously entered lines."

Adding a new sensor or test impacts all states in the state table. Each possible set of conditions must now be considered anew with respect to the new test.

Also, adding state table *lines* (tests) tends to dramatically increase state table *rows* (transitions). If an independent n-valued test is added, the addition of such a new line can cause the number of total states in the state table to increase n-fold. If a dependent test is added, the size of the state table increases by at least the number of extant tests. In each case, the addition of one additional test significantly increases complexity. If a test is added that causes no increase in transitions, it was redundant to begin with.

By representing conditions as in a set of production rules, "don't care" conditions are obviated and the increase in state transitions becomes linear. This technique was used in the HCSE [BBN80] system, however the HCSE shares all of the other faults of state tables.

In conventional programming languages, dependent tests are typically added in a more ad hoc manner. Often a section of code will be rewritten or restructured so that it "makes sense" again. Because of this, the growth of size and complexity is linear and strictly local to the affected area. Independent tests cause the same polynomial growth as in state tables but such tests can be localized from interacting with other tests, thereby significantly diminishing this growth.

> 5) the **if/then** structure of state table lines resembles production systems, and "state tables have all the characteristics of an AI expert system"

This statement indicates a common misunderstanding of AI production systems (which I intend to stamp out right here). Only on the surface are state tables like the if-then rules of expert systems. In such *rule-based* expert systems, the knowledge of the system is encoded in if-then rules which trigger actions. Rules can be weighted according to their relevance. In "programming", an expert congeals a rule or guideline that he has found to be useful ("almost always true"). The rules are sometimes hunches and educated guesses about the way to proceed in problem solving. Expert systems are characterized by "inexact reasoning, using hunches or heuristics to guide and focus what would otherwise be a search of an impossibly large space" [Barr]. In general, such systems are heuristic - they are not guaranteed to give the optimal answer, just a usable one. Rules may be incorrect, contradictory and misleading and yet the system will still work. (This is much the case with the knowledge of most humans.)

In contrast, when a rule from a state table is selected, it must be the correct, best and only rule that is appropriate to the situation. In this case, the programmer must understand the impact of any change or small addition to a rule may have on all other state table rules. When designing or debugging a state table, the programmer must understand the state table in its entirety. One of the original bases for using state tables is that the limited number of states and interactions allowed an easy and total grasp of a state table.

> 6) debugging. "since the set of conditions that lead to and from a state are clearly specified, it is easy to perform traces, set break points and to reason backwards from error states. The system is completely deterministic and errors in logic are possible to reconstruct. Bugs are ... simple to locate and correct."

This brings us to one of the primary disadvantages of state tables. State tables are not "easy to debug". This is primarily because there is *no explicit previous/next state information*. By looking at a state, you cannot tell what the previous state was, nor can you tell what the next will be. For this reason, "reasoning backwards" is extremely difficult. If you explicitly sequence the state table, you may as well use an HLL where sequencing is implicit. In practice, adding new lines during debugging forces one to explicitly reconsider state sequences, whether they are explicit in the table or not. Also, see (3) and (4).

For the same reason, state tables are not easy to read. (They may be clear for "toy" examples using a limited number (i.e. 2 or less) of variables as early examples show.)

Additionally, state tables are thoroughly unexpository as far as displaying algorithms. The tabular notation, complete with its "don't cares" lends no help in expressing the basic components of any algorithm or data structure. As an example, a simple "bounded-loop" requires 4 state transitions. (See figure 2.)

	INPUTS		*OUTPUTS*	
comment	current-state	test	next-state	procedure
initialize	initial-state	don't care	test-state	initialize
test succeeds	test-state	true	increment-state	body
test fails	test-state	false	exit-state	don't care
increment	increment-state	don't care	test-state	increment
exit	exit-state	don't care		

Figure 2 - *bounded loop* rendered as a state table

This is a trivial state table - it has no synchronization, handshaking or other difficult prob-~~lems; it only has two input variables - and yet it is completely opaque! Now imagine a state ta~~ble with 5 inputs and 25 state transitions. Or 9 inputs and 81 state transitions.

Figure 3 shows the same loop rendered as a state graph. The primary difference here is that sequencing, rather than done implicitly or explicitly, is embedded in the representation. A secondary advantage is that "don't cares" are realized as truly redundant and do not appear in the graph. In other ways, however, a state graph representation is not much better than the original state table. (The interested reader may compare this state graph to one directly derived from the state table in figure 2.)

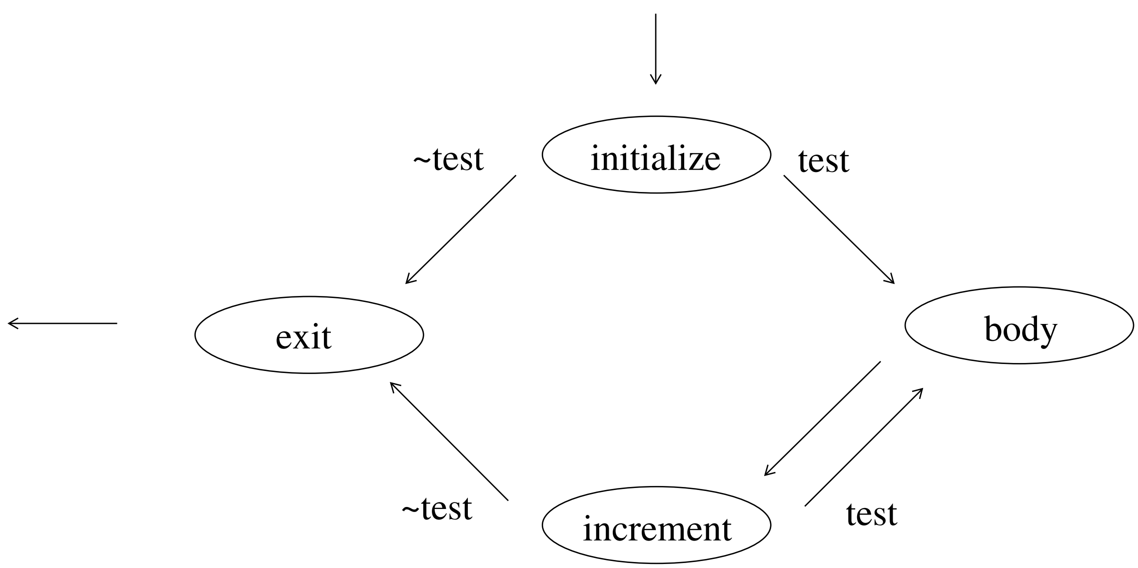

Figure 3 - *bounded loop* rendered as a state graph

Figure 4 shows the same "bounded-loop" in a high-level programming language.

```
for ( initialize test increment ) body
```

Figure 4 - *bounded loop* rendered in a high-level programming language

> 7) "...make it possible to develop programs which learn complex skills incrementally."

The use of the word "learn" brings so many misleading connotations (such as the implication that a system can improve future behavior by studying past experience), it should be used extremely carefully. Suffice it to say, none of the AMRF systems are "learning" systems, nor are they meant to be. I suspect what that was meant was that "programming could be done incrementally". This is the anathema of any structured programming style (top-down, bottom-up, step-wise refinement, etc). This inefficient practice leads to inconsistent programs, having been patched and juried up as each successive run exposes another unthought of problem that must be handled. Such kludged programs will always have "yet another bug" since they are designed with the belief that "we'll fix it when we come to it", rather than "lets consider the structure in toto".

> 8) There are usually "more lines in a state table than are statements in a corresponding procedural program."

This, the sole disadvantage mentioned about state tables, is hardly worth complaining about and I believe the authors knew that. This is an amazingly trivial complaint, especially considering that there are much more substantial ones.

Using "more lines" is perfectly reasonable if there is some gain in ease of representation. However, readability is never mentioned as a benefit of state tables. (Let the record show that I don't know of any truly "readable" HLLs either.) It is possible that some type of graphical representation (such as a state graph) might be much more readable than either a state table of a HLL.

Do state tables have any real advantages? or, What are the alternatives?

It is not clear that state tables have any worthwhile advantages over any decent programming language. Certainly the advantages don't outweigh their disadvantages.

Not really an advantage but worth bringing up is the point that state tables provide direct representations of discrete-time systems, which is what the architecture of the AMRF is designed around to begin with. However, just because we have found an accurate low-level description does not mean that we cannot work with a high-level language and automatically generate (i.e. compile, interpret) low-level code for computer consumption. (E.g. machine code is a very accurate representation of a computer program, but no human communicates with a computer at that level.)

One benefit is the tabular form, which allows clear association between output values and states. Although not as graphically clear, the same effect can be accomplished with message passing semantics in a high-level language. This is an example, of what I mean by a "seductive misadvantage". The tabular form is extremely neat and clean. It is simple to evaluate. There is no language to understand or get in the way as you read or write it. But while these are all true, you suffer exactly for having this simplicity. Its fine for a computer to work in this fashion, but not for a person.

In a state table, predefined variables are bound with state-driven values at each cycle. These values may then be referenced by other processes. With a high-level language, state changes may be declared whenever convenient. A new state with new outputs may be declared by calling "synchronize-state" which would send and receive predeclared (or any, for that matter) variables to and from (for example) a common database. This would update the current process' view of the world and the database (which in turn would update other processes as they performed a "synchronize-state").

In a tightly coupled system, where processes actually share common memory for efficiency (or if there was no message-passing system available), a basic synchronization mechanism (semaphores, monitors, etc.) could be used to allow processes to temporarily lock variables while they were being updated. Thus, a process could enter a critical section in order to perform atomic actions such as updating a set of outputs.

Another benefit is that handling unexpected or erroneous behavior with state tables is potentially not as ad hoc as HLLs. This is because all variables are checked at the beginning of each state cycle. However, this has the drawback of enforcing an alien context-free structuring on the problem. Because of this, the programmer must deal with algorithms on the level of minutiae, while constantly considering how error conditions impact. This point should be driven home by comparing Figures 2 and 4.

This might all be worthwhile if state tables were really capable of handling arbitrary behavior, but in practice the large number of states prohibits all but the simplest type of error-handling.

For example, if an error is detected, it is possible to try and recover from this error although certainly how this is done will depend on the complete state of the process. I claim that this will either never be done or that error recovery will be done extremely simplisticly, primarily due to the number of states that the system could be in. Imagine the state table that had recovery routines for different errors in all the different states.

Trying to produce error detecting and correcting code in state tables also has to handle the problems of simultaneous error handling. Since states have no implied ordering or priority between them, either an ad hoc arbitration scheme must be used in the case of multiple matches, or the programmer must explicitly provide an exponential number of states to handle all the combinations in which that error condition can happen.

An alternative (and what is done in any sequential programming system) is to check all commands for errors (which is often what is done in AMRF state tables, anyway). If problems are encountered, there is knowledge local to that context that can attempt to fix things. Asynchronous conditions can be handled by asynchronous interrupt handlers. Writing an interrupt handler for each type of error is much easier than trying to think of all the possible error states and their combinations that can occur. As opposed to a state table, it is typical that interrupt handlers are prioritized. (Error handlers must be prioritized in state tables as well, but this is done artificially.)

State tables are general enough to implement any algorithm when procedures can be attached to state transitions. Procedures can be used to hide miscellaneous information from the state tables, enhancing readability. However it is not clear what should be hidden versus what should be explicit. In practice, an excessive number of state variables are dealt with by processing them in procedures and vice versa.

Generality is not equivalent to *suitability*. Consistent standards are nice, but, is it reasonable to expect that one formalism is appropriate for the complete range of highest to lowest levels in a control hierarchy (or any two levels for that matter)? The type of control (and thence programming) occurring at each level becomes quite dissimilar as the distance between levels increases. For example, high levels of the AMRF handle problems such as machine scheduling and configuration. Low levels handle operations such as adjusting power to load or stepping a servomotor.

In [Albus81], the hierarchical control of a human is presented. Even though some of this physiology is not well understood, it is apparent that the computational units are quite different in complexity. At the lowest level of the hierarchy, the neuron computes by (weighted) summing of inputs, providing a scalar output. This computational device is capable of performing pattern matching (of which state table evaluation is a subset). At a medium level, the spinal cord is a "formidable computational machine" with a one or two-level structure. Such a one or two-level structure is capable of computing "flight patterns of a bee" and other sophisticated control tasks. At the highest level of the brain, computation is totally symbolic, massively parallel and allows explicitly heuristic and conceptually abstract reasoning. Even though the brain is made up entirely of neurons, these neurons themselves are not used individually except at a very primitive level. At higher levels they are combined to form much more powerful computational machines.

It is true that state tables are capable of describing any type of behavior but only at the cost of added complexity in building constructs that should normally be ignored. For example, iteration, looping, recursion and other tools all require substantial effort to represent in state tables, whereas they are effortless in high-level programming languages and human thought. Is it necessary to force us to deal with all the details at this extremely primitive level?

Conclusion

State tables are a seductive but deficient representation having so many disadvantages and so few advantages that they should be discarded. It is not clear if there is an ideal representation (we are still working on that) but either of state graphs or high-level languages are much easier to work with than state tables. A great deal of work on *automatic programming* and *graphical programming* is currently underway (e.g. [Reiss84]). I expect that such *very-high-level languages* will soon be very important to us and that we should be studying these alternatives now.

The occasional processes or algorithms that are appropriately expressed simply by state tables can still be represented within a high-level language. Certainly, most modern high-level languages have facilities (such as a case statement) to provide state table-like decision processing.

In summary, the primary disadvantages are:

1) State tables force an artificial context-free structuring on processes that aren't.

2) Parallelism and synchronization are hard to represent.

3) Simple algorithms are obtuse when represented using states.

4) Since there is no explicit sequencing, this must either be provided or done ad hoc.

5) Lastly, the number of states in a state table tends to be exponentially related to the number of variables.

It is possible that good implementations of a state table system may allow use of state tables with little pain. For example, one would expect a state-table evaluator to be able to set breakpoints or keep a history of states. However, providing a sophisticated programming environment for state tables is analogous to providing sophisticated tools to minimize logic circuits (i.e. minimize Karnaugh maps) when you should be using microprocessors to begin with. In other words, one would be better off starting with a more powerful or appropriate language to begin with.

References

Albus, J. S., "Brains, Behavior and Robotics", Byte Books, New Hampshire, 1981.

Albus, J. S., Barbera, A., and Fitzgerald, M. L., "Programming a Hierarchical Robot Control System", 12th Int'l Symposium on Industrial Robots, Paris, France, June 1982.

Albus, J. S., McLean, C. R., Barbera A. & Fitzgerald, M. L., "Hierarchical Control for Robots in an Automated Factory", 13th ISIR, Robots 7 Symposium, Chicago, Illinois, April 1983.

Barbera A., Fitzgerald, M. L. and Albus, J. S., "Concepts for a Real-Time Sensory-Interactive Control System Architecture", Proceedings of the 14th Southeastern Symposium on System Theory, April 1982.

Barr and Feigenbaum, E., "The Handbook of Artificial Intelligence, p. 81, Vol. II.

Knuth, D., "The Art of Computer Programming", Vol II, Addison-Wesley, 1984.

Nanzetta, P., "Update: NBS Research Facility Addresses Problems In Set-ups for Small Batch Manufacturing," Industrial Engineering, pp 68-73, June 1984.

Reiss, Steven P., "PECAN: A Program DEvelopment System that Supports Multiple Views," Orlando, FL, March, 1984.

Simpson, J. A., Hocken, R. J., Albus, J. S., "The Automated Manufacturing Research Facility of the National Bureau of Standards, Journal of Manufacturing System, V1, No. 1, 1982.

Winston, P. H., "Artificial Intelligence", pps 132-136, Addison-Wesley, 1984.