LOGIC SIMULATION USING NETWORKS OF STATE MACHINES*

Peter M. Maurer

Department of Computer Science & Engineering
University of South Florida
Tampa, FL 33620

Abstract

This paper shows how to simulate a circuit as an interlocked collection of state machines. Separate state-machines are used to represent nets and gates. The technique permits intermixing of logic models, direct simulation of higher-level functions, and optimization techniques for fanout free circuits. These techniques are shown to be an extension of the Inversion Algorithm, a high-performance event-driven simulation technique. New, more efficient state-machine implementations are presented, and experimental data is presented that show the efficiency of the new techniques.

* This research was supported in part by the National Science Foundation under grant MIP-9403414.

LOGIC SIMULATION USING NETWORKS OF STATE MACHINES

Peter M. Maurer

Department of Computer Science & Engineering
University of South Florida
Tampa, FL 33620

1 Introduction.

It is a common practice to view a digital circuit as a network of components with signals flowing through it. For example, a carry-lookahead adder can be viewed as a network of gates with operands flowing in, undergoing transformations, and flowing out as a combined sum. Although this view is useful, it does not reflect reality. In reality an electrical component is a device with an internal state which changes in response to purely local conditions. Signals do indeed flow along conductors from one device to another, but it is the change in state of the conductor at the target device that causes switching activity. Signal flow is primarily a means of propagating local state-changes from one component to another.

At first it seems to be splitting hairs to make this distinction between global signal flows and local state changes. However, this change in viewpoint is actually quite useful. For example, it is no longer necessary to view an AND gate as a device for computing a specific Boolean function. Instead, one can view the AND gate as a device with an internal state that changes in response to changes in its inputs. As shown in [1], this makes it possible to compute the state of a multiple-input AND gate using unary operations. Since Boolean functions are no longer being computed, it is no longer necessary to use ones and zeros to represent the state of a net. Any convenient representation will suffice.

To illustrate the utility of this approach, consider the network of state machines illustrated in Figure 1. This is a state-machine representation of a three-input AND gate, although an identical structure could be used for NAND, OR, and NOR gates.

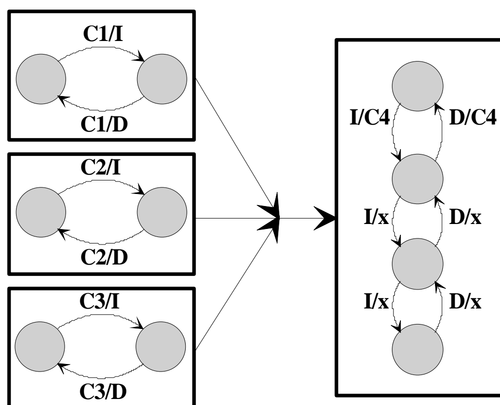


Figure 1. A Sample Network.

Figure 1 illustrates the terminology used in this paper, as well as the general approach to networking state machines. Each rectangle represents a single state machine, and the symbols on the arcs represent inputs and outputs of the state machine. The symbol C1/I indicates that on input C1, the associated state transition will occur, and the state machine will produce the output I. Because the outputs are associated with state transitions, these are Mealy machines rather than Moore machines. In many cases it is convenient to associate an additional output with each state. When this is done, the outputs associated with transitions will be called the *Mealy outputs*, and the outputs associated with states will be called the *Moore outputs*. State machines interact with one another via their Mealy outputs, with the Mealy outputs of one machine will be the inputs of the following machine.

In Figure 1, the three state machines on the left represent the three inputs to the AND gate, while the state machine on the left represents the gate itself. The symbols chosen for the inputs and outputs are taken from the Inversion Algorithm implementation of these machines. The symbols C1, C2, C3, and C4 represent changes in signals 1 through 4 respectively. In the Inversion Algorithm this change is represented by queuing and processing an event. A count of dominant inputs is kept for each AND gate, and the symbols I and D represent incrementing and decrementing the count. The count itself is represented by the state of the machine on the left. The top state represents zero. Transitions into and out of this state will cause events to be propagated. The other transitions don't generate any output. The lack of a Mealy output is represented by the symbol ***x***.

The advantage of using networked state machines such as that illustrated in Figure 1 is that it is not necessary to completely re-evaluate the AND function when an input changes value. All computations are local and done on single states. Unlike more typical simulation, adding an input does not increase the amount of work that must be done to process the existing inputs. Furthermore, state transitions can be made without regard to net values. Although the three state machines on the left of Figure 1 have two states, it is not specified which state corresponds to zero. In fact, for an AND gate, the rightmost state of each machine would correspond to zero. If we were to designate the leftmost state as the zero state, the structure of Figure 1 would then correspond to an OR gate instead of an AND. *No change to the structure would be required!* By mixing and matching the designations between right and left, still other functions could be implemented.

Despite the notation, it is important to emphasize that the Inversion Algorithm implementation is only one implementation of the structure of Figure 1. There are many other implementations for this network of state machines, and in many of these implementations, I and D will represent something other than Increment or Decrement.

The purpose of this paper is to explore the concept of networked state machines more thoroughly. (A similar, but fundamentally different approach is discussed in [5].) It will be shown that there are more effective implementations than those used by existing simulation algorithms, and that it is possible to represent complex Boolean functions as simple state machines. The efficiency of this approach to will be demonstrated using experimental data.

2 State Machines.

The most fundamental state machines are those used to represent nets. A state machine for a two-valued net is illustrated in Figure 2. This state machine has a single input called either "Event" or "Change." More complex state machines, such as that illustrated in Figure 3, can be used to represent more complex logic models. (In Figure 3, the transitions into and out of the unknown state are not fully specified.) Since a state machine is a local entity, it is possible to use different logic models for different nets. (This concept is explored briefly in [2].) To simplify the discussion, this paper will assume that all nets will be represented using a state machine similar to that of Figure 2.

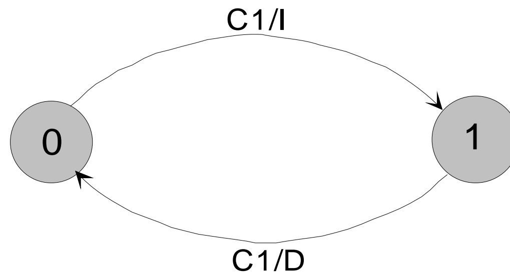


Figure 2. A Binary State Machine.

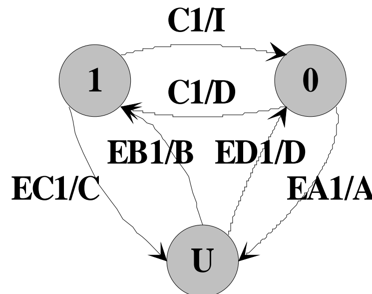


Figure 3. A Trinary State Machine.

In the state machine of Figure 2, the Moore outputs 0 and 1 represent the values of the net. These values are required only during the initialization phase of the simulation, and are not present during normal simulation. Because these outputs are "virtual" the assignment of ones and zeros can be switched whenever it is convenient to do so. This

will have no effect on the progress of the simulation, but will alter the function computed by the state machines.

Figure 4 illustrates the system of interconnected state machines for a two-input AND gate with all Moore outputs specified.

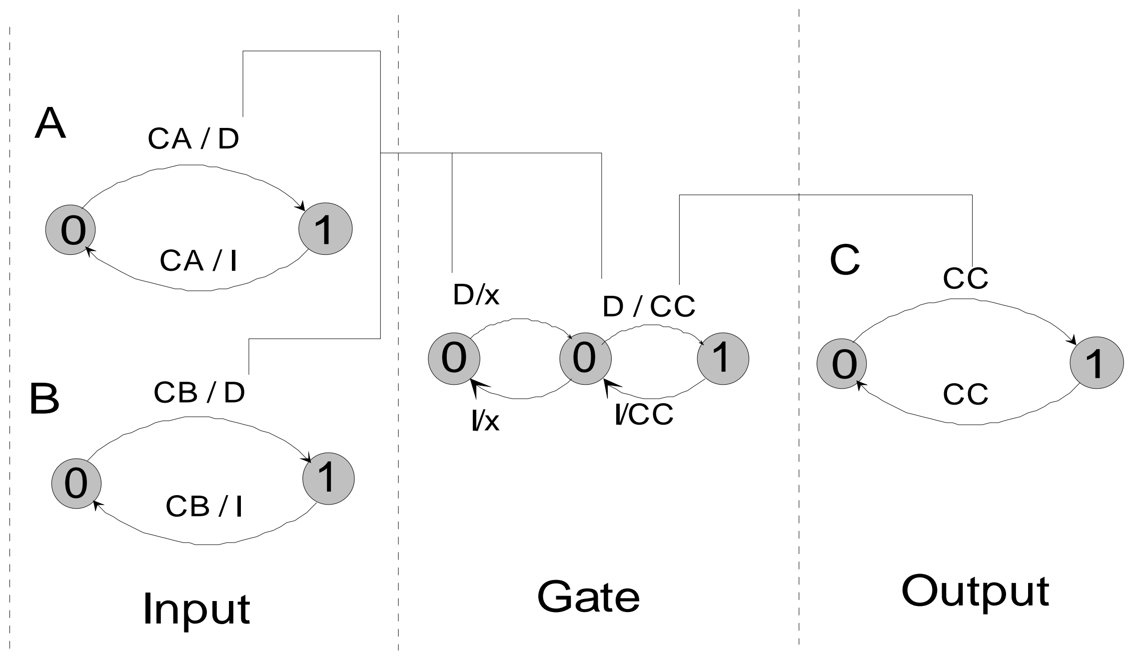


Figure 4. State Machines for an AND gate.

By switching the Moore outputs of the input and output state machines, it is possible to simulate many different types of gates using the same system of state machines. Figure 5 illustrates the functions that can be computed using the structure of Figure 4. The A, B, and C columns indicate the Moore output of the leftmost state. It is possible to implement even more functions by altering the Mealy outputs of the Gate state machine, as illustrated in Figure 6.

A	B	C	Function
0	0	0	*A* **AND** *B*
0	0	1	*A* **NAND** *B*
0	1	0	*A* **AND Not** *B*
0	1	1	**Not** *A* **OR** *B*
1	0	0	**Not** *A* **AND** *B*
1	0	1	*A* **OR Not** *B*
1	1	0	*A* **NOR** *B*
1	1	1	*A* **OR** *B*

Figure 5. Functions Computable With A Linear State Machine.

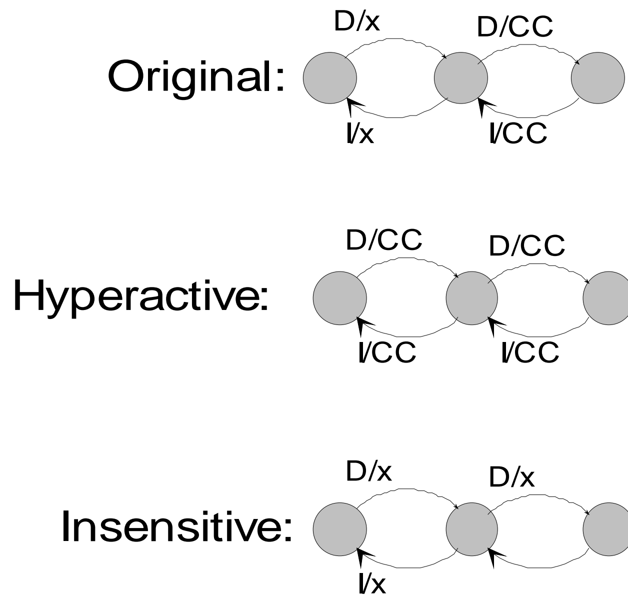


Figure 6. Three Types of Gate State Machine.

The Hyperactive machine can be used to simulate the XOR and XNOR functions, while the Insensitive machine can be used to simulate constant one and constant zero. These functions can also be simulated by less elaborate mechanisms.

Most state machines used by the Inversion Algorithm are linear in shape, because they correspond to a count of dominant inputs. Although limited in structure, these machines can be quite powerful, especially for functions with more than two inputs. Figure 7 illustrates the state machines used by 3 and 4-input AND gates.

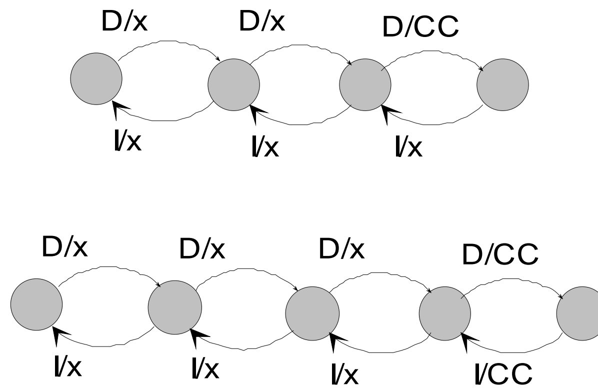


Figure 7. Three and Four-Input Linear State Machines.

As with 2-input gates, the input and output state machines can be modified, to simulate many different functions. However, because there are more than two transition points, the structure of the Mealy outputs can be correspondingly more complex. Figure 8 gives the variations of the 3-input linear state machine.

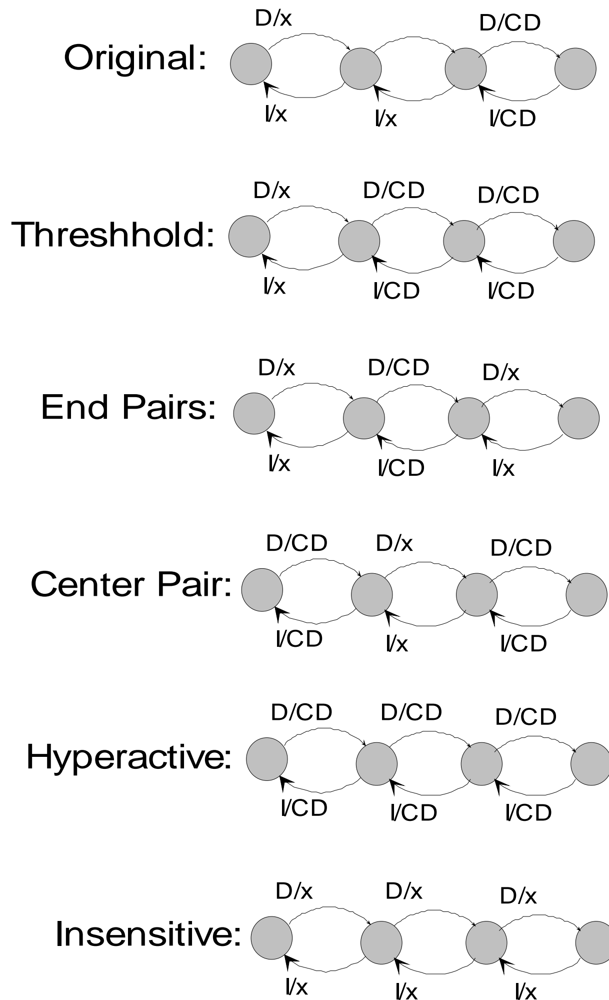


Figure 8. Types of 3-Input Gate State Machines.

Despite the wide variety of functions that can be simulated using linear state machines, for $n>2$, it is impossible to implement all n-input functions as linear state machines. A simple counting argument shows that this is true. For n inputs, the linear state machine will have $n+1$ states, and n links. Altering the input and output state machines will give no more than 2^{n+1} different functions per state machine, while altering the Mealy outputs of the gate state machine will yield no more than 2^n different state machines. This gives a total of 2^{2n+1} different functions. However, there are 2^{2^n} different n-input functions. This implies that direct simulation of certain functions will require a wider variety of state machines. As an example of such a machine, consider the cube-shaped machine in Figure 9. This machine can be used to simulate any 3-input function. For n inputs, the corresponding state machine will be an n-dimensional hypercube with 2^n states. The state machine of Figure 9 is the cross product of three two-state machines, while the general cubic state machine is the cross product of n such machines. It is also possible to form the cross product of other types of linear state machines, as illustrated in Figure 10.

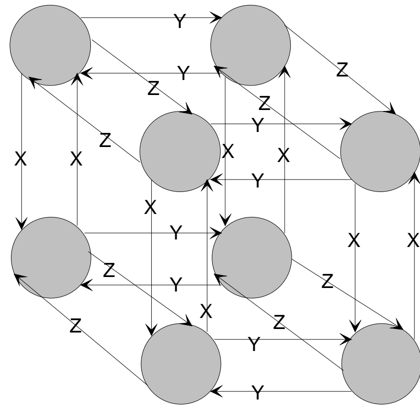


Figure 9. A 3-Input Cubic State Machine.

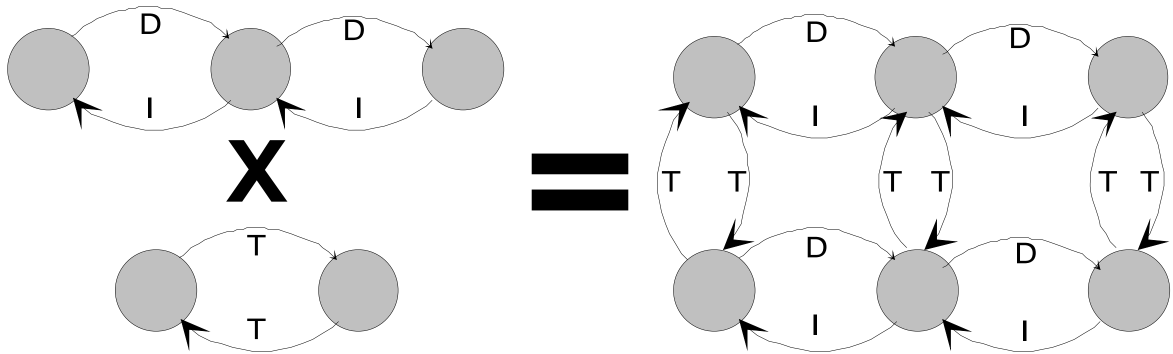


Figure 10. The Cross Product of Two Linear State Machines.

3 Systematic Generation of State Machines

A state machine for a complex function can be derived from the basic state machine of Figure 2. The general procedure is to create a cross-product machine, assign Moore outputs based on the function, and then reduce the machine using standard techniques. To illustrate the procedure, we will demonstrate how to create a state-machine for a two-input AND. We start with the fundamental state machines for the inputs, and take the cross-product of these machines as shown in Figure 11.

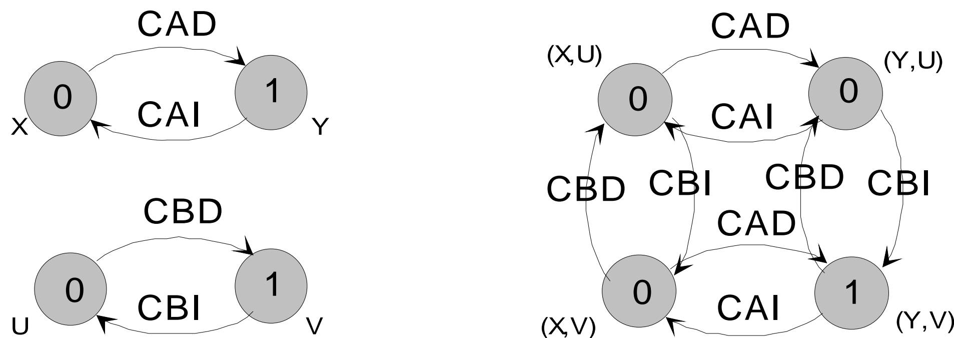


Figure 11. Developing a Gate State Machine.

The Moore outputs of each state are computed by applying the AND function to the Moore outputs of the original machine. The Mealy outputs are computed by identifying

the transitions that cause the Moore output of the machine to change. For more complex functions, the computation of the Moore outputs would use the function itself, and the computation of the Mealy outputs would proceed as before by identifying transitions where the Moore output changes.

To simplify the resulting state machine, it is to identify symmetric inputs, leading to some linearization of the machine. In Figure 11, the symmetric inputs A and B can be collapsed by replacing the inputs CAI and CBI with a single input I, and the inputs CAD and CBD with a single input D. This will yield the non-deterministic state machine illustrated in Figure 12. The general procedure for handling symmetric input nets is to combine the associated state machine inputs with a single input representing a change in any of the input nets.

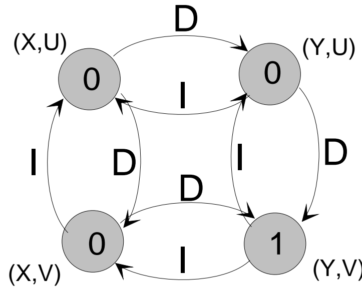


Figure 12. Combined Inputs.

Before the state machine of Figure 12 can be implemented, it is necessary to convert it into a deterministic machine. This will eventually produce a state of the form {(X,V),(Y,V),…}. The equivalent deterministic state machine is given in Figure 13.

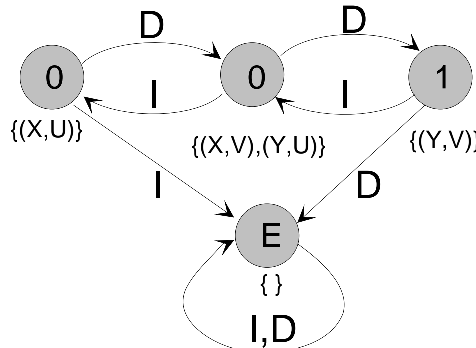


Figure 13. The Equivalent Deterministic Machine.

Note that, except for the Error State, the state machine of Figure 13 is precisely the gate state machine illustrated in Figure 4. The reason the Error State was missing from Figure 4, is that the transitions to that state cannot occur. (In the remainder of the paper, we will omit the error state when no transition into that state can occur.)

Although a the gate state machine for a function can always be created starting with simple binary machines, it is sometimes easier to treat the function as a collection of simpler functions. This is particularly true when collapsing homogeneous and heterogeneous connections as described in [1]. For illustrative purposes, assume we wish to find the gate function for a gate consisting of three gates G, H, and K, connected as illustrated in Figure 14. (The number of inputs to either G or H is unimportant.)

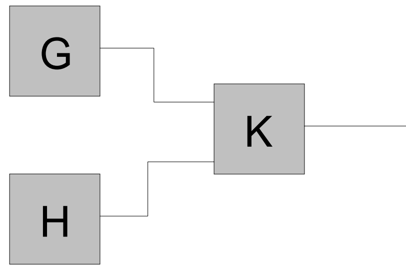


Figure 14. A Collection of Gates.

Assume that the gates G, H, and K are connected as illustrated in Figure 14, and suppose the function computed by gate K is z=f(x,y). Taking the cross product of the state machines for G and H creates the state machine for the combined gate. The Moore output of the state (P,Q) is f(p,q), where p is the Moore output of state P and q is the Moore output of state Q. It may be possible to combine symmetric inputs in the result. Figure 15 illustrates this process on two homogeneous connections.

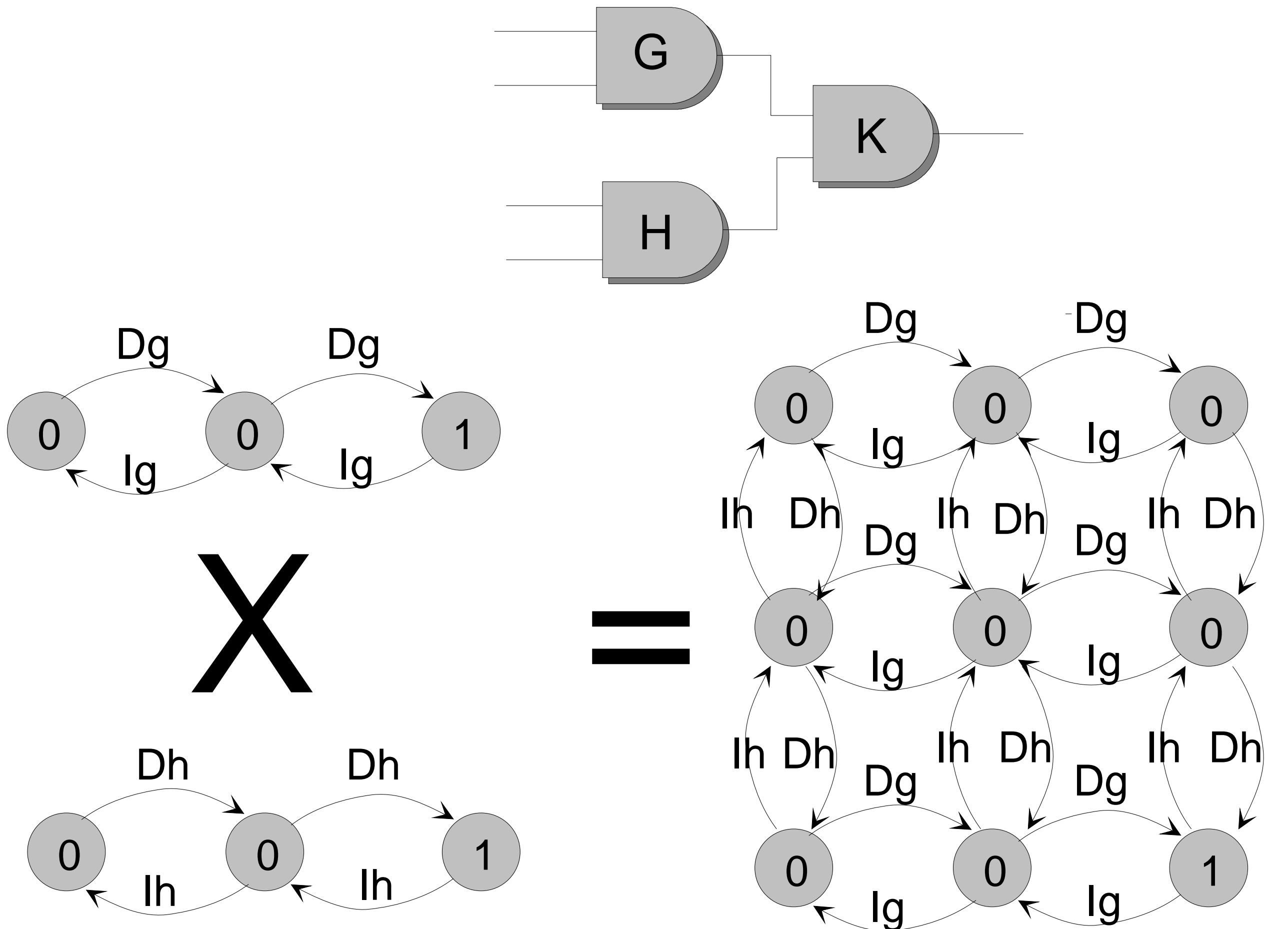


Figure 15. Using State Machines to Collapse Homogeneous Connections.

In Figure 15, it is possible to combine symmetric inputs using the rules Dh = Dg = D, and Ih = Ig = I. This will yield a non-deterministic state machine which will then yield the deterministic state machine of Figure 16.

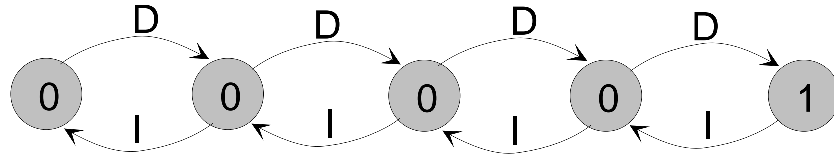


Figure 16. The Deterministic Equivalent.

The same procedure can be used to collapse heterogeneous connections, as illustrated in Figure 17. The results are less elegant than those for homogeneous connections, but the resultant code is more compact than that produced by the layered method of collapsing homogeneous connections. (See [1].)

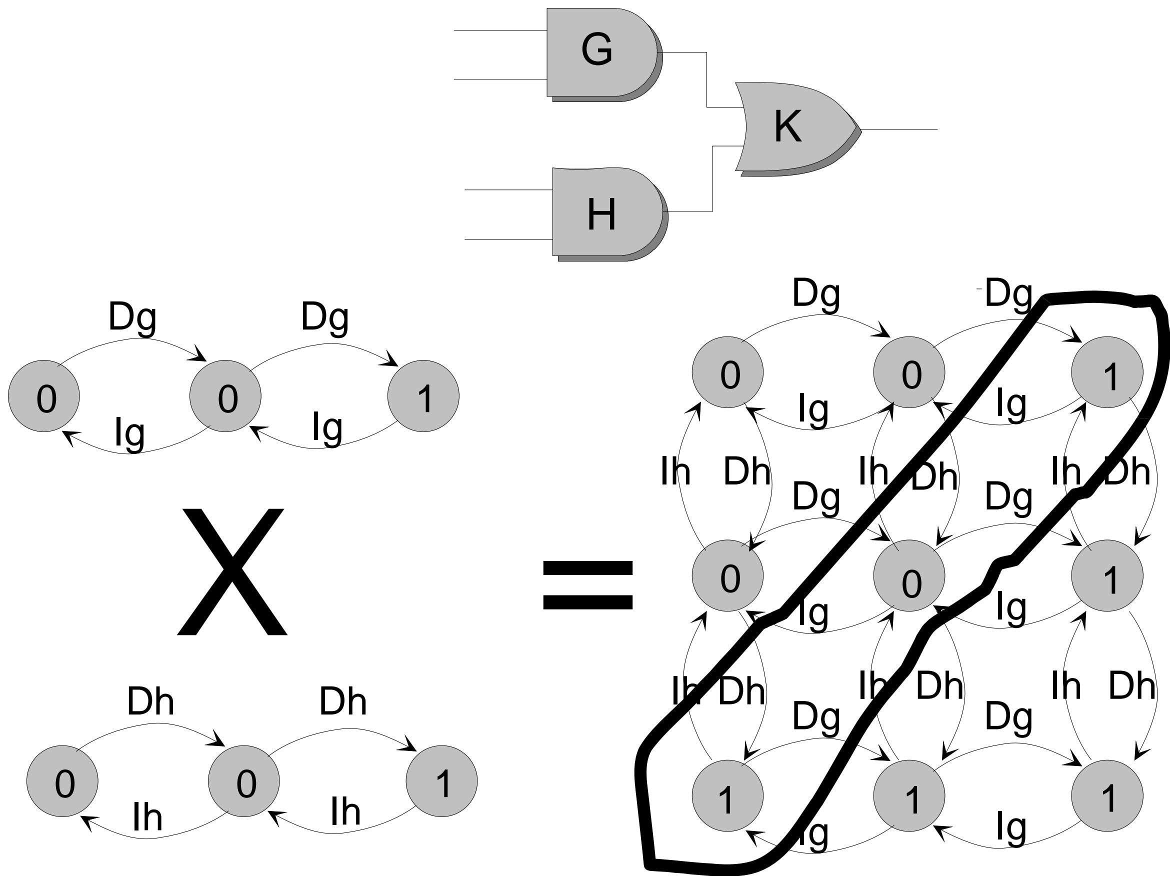


Figure 17. Collapsing Heterogeneous Connections Using State Machines.

The state machine of Figure 17 cannot be collapsed into a linear state machine because this would require combining the three circled states into a single state. Since these states have different Moore outputs, this would destroy the function of the machine.

After constructing the combined state machine, and possibly reducing it, it is necessary to assign the Mealy inputs. Any transition between states with identical Moore outputs is assigned a Mealy output of NULL. All other transitions are assigned Mealy

outputs of EvX, where X is the output of the gate. Once the Mealy outputs have been assigned, the Moore outputs can be dropped. It is possible to implement the resultant state machine in many different ways. The next section gives some preferred implementations.

4 Implementation of State Machines

In the Inversion Algorithm, all state machines are implemented by the event processing routines of the various nets. The binary transitions of the nets are modeled by alternating two different event handlers, which produce the D and I outputs respectively. Each routine is responsible for scheduling its counterpart. The precise mechanism for is described in [1]. This model can be extended to handle the more complex state machines described in this paper.

As in the Inversion Algorithm, it is possible to embed all state machine implementations in the event-handlers for various nets. The primary function of an event processing routine is to handle the state transition and event propagation of a gate state machine. The most obvious way to handle linear state machines is to maintain a count, and use the I input to increment the count and the D input to decrement the count. Event propagation can be done by examining the count after a change. For the standard state machines illustrated in Figure 4 and Figure 7, the count is tested one after an increment and zero after a decrement. Although this procedure has given good for results for simple linear state machines, there are many other techniques that could be used. For more complex state machines, more sophisticated procedures must be used.

Consider a state machine M, which is the cross product of two linear state machines A and B. Assume that the inputs to A are I_a and D_a, and the inputs to B are I_b and D_b. Assume that machine A is implemented using a count, and that k is the maximum value of that count, and that the count can never become negative. In the machine M, the inputs I_a and D_a are assigned the value 1, and the inputs I_b and D_b are assigned the value k+1. The machine M will be implemented using a single count. The input I_a will increment the count by 1, the input D_a will decrement the count by 1, the input I_b will increment the count by k+1, and the D_b input will decremented the count by k+1. This principle is illustrated in Figure 18, which shows the state machine of Figure 17 with increment and decrement values added.

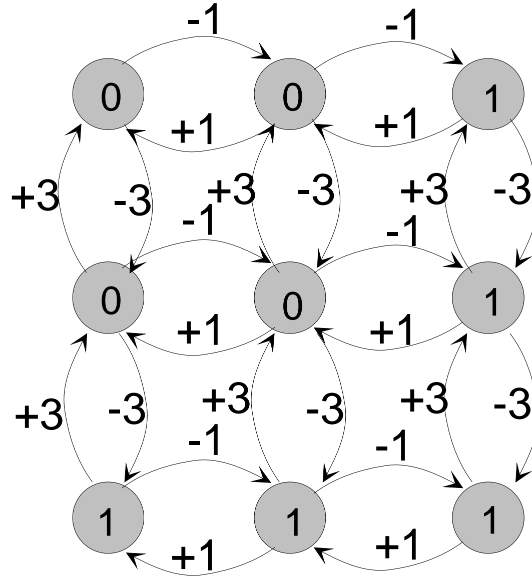


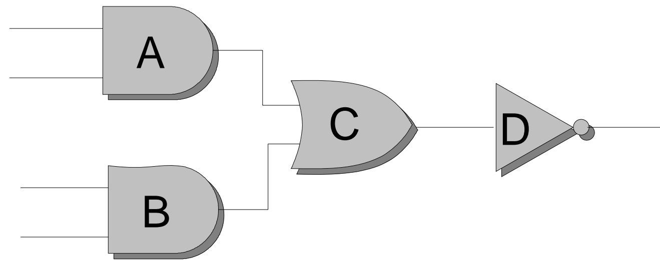
Figure 18. Cross-Product Machine with Increments and Decrements.

Figure 18 also shows how to determine when to propagate events. After an increment by 3, an event will propagate if the new count is 4 or 5. After an increment by 1, an event will propagate if the new count is 4 or 7. After a decrement of 3, an event propagates if the new count is 1 or 2, and after a decrement of 1, an event propagates if the new count is 3 or 6.

Other cross-product machines can be implemented in a similar way. A *general counting machine* is defined to be either a linear state machine, or a cross-product of two other general counting machines. All general counting machines can be implemented using a single count, which is incremented and decremented by different amounts depending on the input. Suppose M is the cross product of two general counting machines A and B. As in the previous example, the inputs of A are assigned the same value that they had in the implementation of the machine A. If k is the maximum value of the count in the implementation of machine A, then the inputs of B are assigned the value $(k+1)q$ where q is the value of the input in the implementation of B. Cross products of more than 2 general counting machines can be created by forming cross products two at a time. That is, A×B×C = (A×B)×C. Cubic state machines such as that illustrated in Figure 9 are general counting machines, so general counting machines are powerful enough to simulate any Boolean function.

Although incrementing and decrementing counts is easy, testing the new count for event propagation can be quite complicated. Instead of comparing the new count against a single value, it may be necessary to compare against a large list of values. This can add significantly to event-processing time. However, both the increment process and the testing process can be simplified by using multiplication instead of addition as the state-transition function. Instead of using counter-states of 0, 1, 2, 3, … one uses states of the form 2^0, 2^1, 2^2, 2^3, … . Instead of incrementing and decrementing by k, the count is multiplied or divided by the 2^k. Since these operations are performed only on positive numbers, which are themselves powers of 2, left and right shifts can be used to perform the multiplication and division. Since each bit position in the counter represents a different state, it is possible to use a mask to test for several states simultaneously.

Suppose it is necessary to determine whether machine M is in state *s*, where *s* is any element of the set **S**. The mask for performing this test will contain ones in those bit-positions that correspond to elements of **S**, and zeros elsewhere. The test is performed by ANDing the mask with the state counter. A non-zero result indicates that M is in some state $s \in \mathbf{S}$. As illustrated in Figure 17, the state machine for the AOI22 gate has nine states, and cannot be reduced by combining inputs. Figure 19 shows the implementation of this state machine using shifting and masking.



IA Transitions.
Count = Count << 1;
if ((Count & 0x24) != 0)
{
Queue Following Event;
}
SP = SP->Next;
goto *SP->RTN;
	IB Transactions:
	Count = Count << 3;
	if ((Count & 0xC0) != 0)
	{
	Queue Following Event;
	}
	SP = SP->Next;
	goto *SP->RTN;

DA Transitions:
Count = Count >> 1;
if ((Count & 0x12) != 0)
{
Queue Following Event;
}
SP = SP->Next;
goto *SP->RTN;
	DB Transactions:
	Count = Count >> 3;
	if ((Count & 0x18) != 0)
	{
	Queue Following Event;
	}
	SP = SP->Next;
	goto *SP->RTN;

Figure 19. The Implementation of A Cross-Product Machine.

The implementation shown in Figure 19 is significantly simpler than the layered implementation presented in [1]. Each bit of the variable "Count" represents a different state. Only the nine low-order bits of "Count" are significant. Figure 20 shows the correspondence between bit positions and the Moore outputs of the state machine.

State Number	8	7	6	5	4	3	2	1	0
Bit Position									
Moore Output	0	0	0	0	1	1	0	1	1

Figure 20. State Counter for the AOI22 Gate.

Figure 20 can be used to construct the masks used in Figure 19. The I_a and D_a transitions cause right and left movement within a 3-bit group, while the I_b and D_b transitions cause movement between three-bit groups. In either case, only movement between immediate neighbors is permitted. The two I_a transitions that cause event propagation are the transitions 4->5 and 1->2, while the two D_b transitions that have cause propagation are 7->4 and 6->3. The binary masks for these two types of transitions are 000,100,100 and 000,011,000.

5 Experimental Data

Although it is possible to use the techniques of this paper to automatically optimize circuits, it is also possible to provide function libraries of common functions, which can be simulated as single gates. To illustrate this process, we have constructed several circuits using conventional gates. After simulating these circuits using the Inversion Algorithm, we replaced various common subcircuits with functions that could be simulated as a single gate. In the first experiment, we constructed a 64x64 array multiplier using half- and full-adders. The half- and full-adder circuits are illustrated in Figure 21.

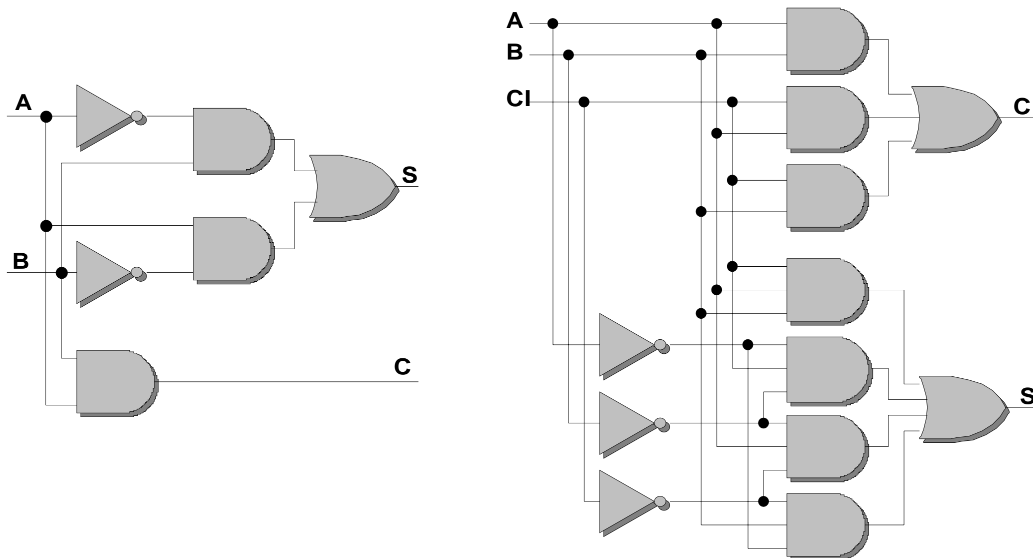


Figure 21. Half and Full Adders.

We constructed a second 64x64 array multiplier that used single-gate functions for the carry and sum. Simple XOR functions can be used for the sum, but it is necessary to create a custom function to compute the carry for the full-adder. This function can be computed using the "End Pairs" state machine of Figure 8. (In the implementation, the

function is treated as an OR gate, with the count initialized to a negative one instead of zero.) To measure the performance of these two circuits, they were simulated on a SUN 300MHz single processor Ultra SPARC-II with 128MB of RAM. Five thousand randomly generated input vectors were used to perform the simulation. The results are reported in Figure 22 in seconds of CPU time. As is to be expected, the implementation using functions is significantly faster.

Implementation	CPU Sec.
Gates	**74.0**
Functions	**15.9**

Figure 22. Simulation Times for 64x64 Multipilers.

We conducted a similar experiment using carry-lookahead adders of various sizes. These circuits were constructed from 4-bit carry-lookahead units arranged hierarchically. The functions computed by these units are given in xxx. It is assumed that the propagate and generate functions have already been computed. To support the hierarchical structure, each unit computes three carries, and group propagate and generate functions. Each unit has four propagate and four generate inputs, as well as a carry-in, labeled C_0.

$$C_1=P_1C_0+G_1$$
$$C_2=P_2P_1C_0+P_2G_1+G_2$$
$$C_3=P_3P_2P_1C_0+P_3P_2G_1+P_3G_2+G_3$$
$$GP=P_1P_2P_3P_4$$
$$GG=P_4P_3P_2G_1+P_4P_3G_2+P_4G_3+G_4$$

Figure 23. Carry Lookahead Unit Equations.

Several carry-lookahead adders were created using these units. Two different implementations were used, one which used conventional AND and OR gates to implement the functions, and the other that used a single-function implementation of each equation. The single-function implementation of these equations demonstrates the utility of the custom-function approach. If the function for C_3 were created automatically, a 10-input function would be created. Using a custom implementation, it is possible to create a seven-input function. The resultant function is much simpler than would be created by automatic methods. To be specific, in the implementation of the C_3 (and GG) function, inputs P_1 and C_0 are assigned the value 1, inputs P_2 and G_1 are assigned the value 2, inputs P_3 and G_2 are assigned the value 4, and the input G_3 is assigned the value 8. An event is propagated when the count goes from less than 8, to greater than or equal to 8, or vice versa.

The results of the carry-lookahead adder experiments are reported in CPU seconds in Figure 24. Five thousand randomly generated vectors were used for each circuit. Because the function implementation eliminates a lower percentage of gates, the performance gains, though substantial, are smaller than those for the array multiplier.

Size	Gate Implementation	Function Implementation
4x4	0.3	0.2
8x8	0.6	0.5
16x16	1.2	0.9
32x32	3.0	2.0
64x64	7.3	5.3
128x128	16.1	13.6
256x256	31.8	24.4
512x512	85.8	57.4

Figure 24. Carry-Lookahead Adder Performance.

6 Conclusions

The state-machine analysis presented in this paper can significantly increase the simulation power and speed of gate-level simulation. The techniques described here allow many different types of functions to be simulated as a single gate. The techniques are based on the same concepts as the Inversion Algorithm but are not limited to a small class of symmetric functions. Although the Inversion Algorithm was able to collapse heterogeneous (AND-OR) connections, only a portion of the simulation code could be eliminated. This paper presents a much simpler method for handling such connections, and opens the door to additional research on the simulation of circuits using state machines.

It is not known to what degree partial and total symmetry will help in reducing state machine size. This is due to the unique view of symmetry taken by the state machine approach. Because of the elimination of net values, it is possible to remove all NOT gates from a simulation. This means that many functions that would normally be considered non-symmetric can be treated as if they were totally symmetric. In fact, Section 2 makes it clear that in the state-machine approach, *all* 2-input Boolean functions are symmetric.

It is clear from Figure 17 that there are non-trivial symmetry classes other than those represented by the linear state machines. As yet, it is not known what symmetry classes exist for a given number of inputs, and how extensive these classes might be. In particular, it is not known whether a full cubic state machine is required for *any* function, and if such functions exist, what the minimum number of inputs is necessary to impose such a requirement.

The techniques presented here can be used to improve the performance of logic simulations. The problems described here will serve as the basis of much future research in improving the performance of digital logic simulation.

7 References

1. P. M. Maurer, "The Inversion Algorithm for Digital Simulation," Proceedings of ICCAD-94, pp. 259-61.
2. P. M. Maurer, W. J. Schilp, "The Three-Valued Inversion Algorithm," Submitted for Publication. Available from the author (maurer@csee.usf.edu).

3. W. J. Schilp, P. M. Maurer, “Unit Delay Simulation with the Inversion Algorithm,” Proceedings of ICCAD-96, pp. 412-7.
4. D. Schuler “Simulation of NAND Logic,” Proceedings of COMPCON 72, Sept 1972, pp. 243-5.
5. M. Heydemann, D. Dure, “The Logic Automation Approach to Accurate Gate and Functional Level Simulation,” Proceedings of ICCAD-88, pp. 250-253.