

*Logic Simulation  
with Interlocked  
State Machines*

**Peter M. Maurer**

Technical Report DA-33

VCAPP Laboratory

Dept. of Computer Sci. & Eng.

University of South Florida

Tampa, Florida 33620

# LOGIC SIMULATION USING A COLLECTION OF INTERLOCKED STATE MACHINES\*

**Peter M. Maurer**  
**William J. Schilp**

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

## Abstract

Most logic simulation proceeds in a straightforward manner with the code of the simulator performing the same logic operations as the gates of the circuit being modeled. Explicit structures are used to represent the nets and the gates of the circuit. Recent work has shown that a more mathematical and less intuitive approach to logic simulation can give substantial benefits both in simulation time, and the amount of memory required to perform a simulation. Specifically, the Inversion Algorithm represents a circuit as a collection of interlocked state machines. Net values are not used, and a substantial portion of the circuit can simply be ignored without affecting the results of the simulation. Despite the impressive results that can be obtained with the Inversion Algorithm, its state-machine models are quite primitive. This paper begins by analyzing the Inversion Algorithm and its state machine models. The theoretical foundations of the algorithm are explored, and more extensive state machine models are presented. Recommendations are made for using these results to construct a logic simulator. These results are not aimed at a particular simulator, but are designed to be used in many different type of simulators. This work is also intended to serve as a basis for more extensive work in the modeling of complex circuits.

---

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

# LOGIC SIMULATION USING A COLLECTION OF INTERLOCKED STATE MACHINES\*

**Peter M. Maurer**  
**William J. Schilp**

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

## 1 Introduction.

The Inversion Algorithm [1-4] is an unconventional event-driven simulation algorithm whose performance rivals that of Levelized Compiled Code simulation. The

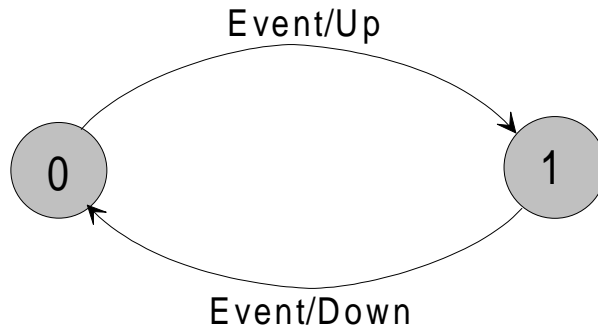
The original description of the Inversion Algorithm [1] focuses on the mechanisms used to simulate simple gates, without exploring the theory upon which these mechanisms are based. The two mechanisms that are described in [1] are counting, and transparent propagation of events. Some reasonably elaborate counting mechanisms are used to heterogeneous connections. (The simplest form of heterogeneous connection is a connection between an AND gate and an OR gate. A simple homogeneous connection is a connection between two OR gates.) Curiously enough, counting is incidental to the design of the Inversion Algorithm. The intent of the algorithm was, and is, to prevent any gate from being simulated unless its output is guaranteed to change value. Counting is merely a convenient method for achieving this goal. The purpose of this paper is to provide some of the theoretical background of the Inversion Algorithm, and to use this theory to extend the simulation power of the algorithm.

## 2 State Machines.

The focus on mechanisms rather than on theory was necessary to simplify the presentation of the algorithm and its workings. However, this approach also tends to obscure both the theoretical underpinnings of the algorithm, and some of its more obvious extensions. Although the Inversion Algorithm may appear to be a collection of counting techniques, a more accurate view is to look at it as a technique for transforming a circuit into an interconnected network of finite state machines. (A similar, but fundamentally different approach is discussed in [5].) The most fundamental of these machines is that illustrated in Figure 1, which is used to represent nets. This state machine has a single input called either "Event" or "Change."

---

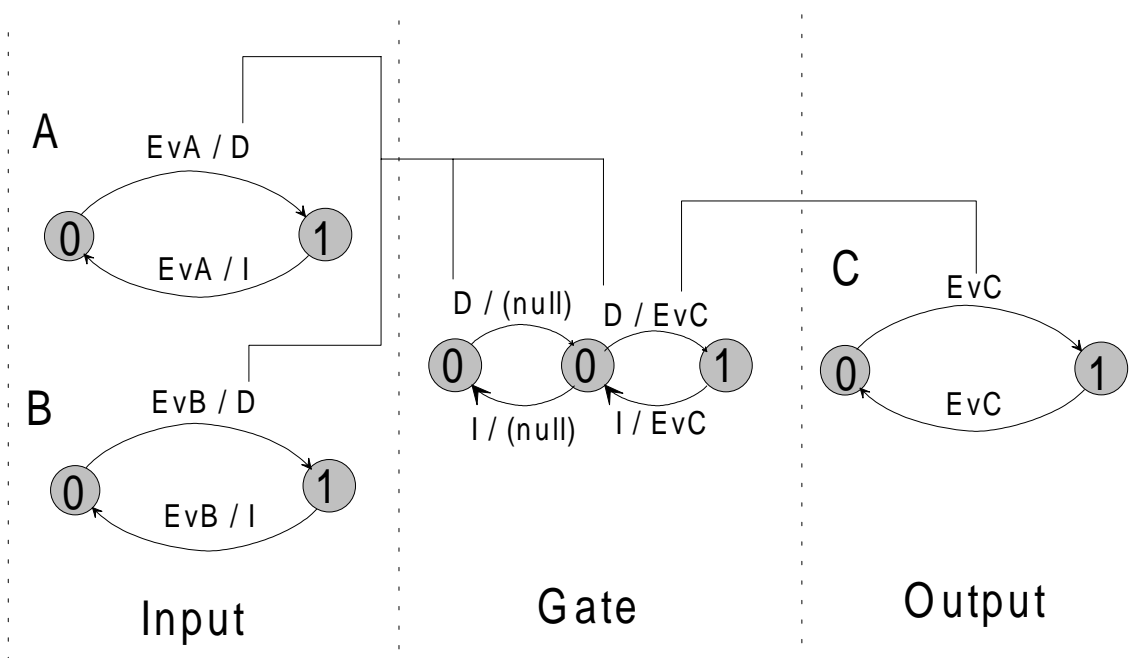
\* This research was supported in part by the National Science Foundation under grant MIP-9403414.



**Figure 1. A Binary State Machine.**

In the state machine of Figure 1, the outputs 0 and 1 represent the values of the net. The interconnection between state machines depends on additional outputs that are associated not with states, but with state transitions. These are the outputs Up and Down illustrated in Figure 1. The outputs that are associated with states are called the *Moore* outputs of the state machine, while those associated with transitions are called the *Mealy* outputs of the machine. Moore outputs are used to represent net values, and are required only during the initialization phase of the simulation.

Figure 2 illustrates the system of interconnected state machines that would be used to simulate a two-input AND gate.



**Figure 2. State Machines for an AND gate.**

In Figure 2, the two state machines on the left represent the inputs of the gate. Events (EvA and EvB) on the inputs generate I and D events for the center state machine, which represents the state of the gate. The center state machine generates EvC events for the gate output. Most of the unique properties of the Inversion Algorithm grow out of the state-machine structure. Furthermore, more complex state machines can be used as gate

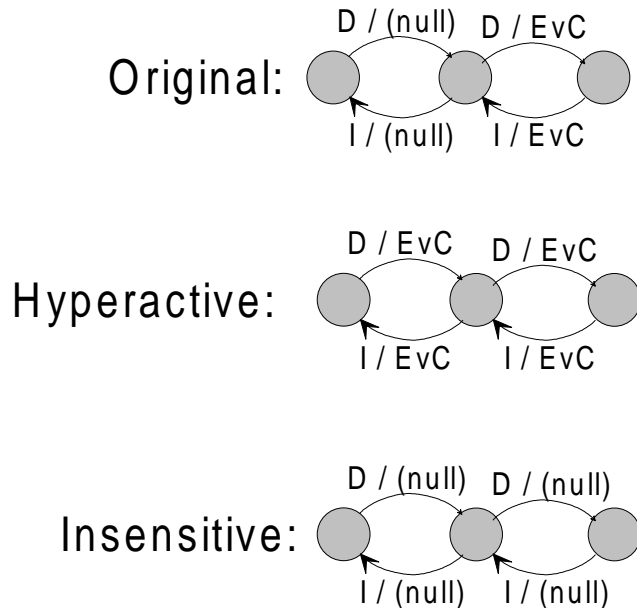
state machines, which can enhance both the simulation power of the algorithm and its efficiency.

As Figure 2 illustrates, the Moore outputs do not affect the progress of the simulation. 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. This corresponds to the elimination of NOT gates described in [1]. Figure 3 illustrates the functions that can be computed using the state machine structure of Figure 2. The A, B, and C columns indicate the Moore output of the leftmost state. The rightmost state of the same machine is assumed to have the opposite value. The left-to-right transition will produce a D output, while the right-to-left transition will produce an I output.

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 3. Functions Computable With A Linear State Machine.**

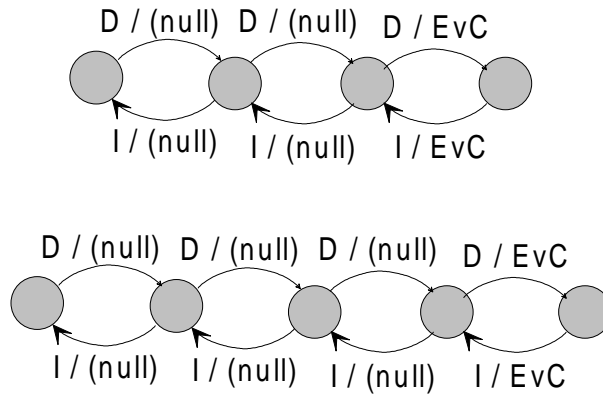
It is possible to implement even more functions by altering the Mealy outputs of the Gate state machine, as illustrated in Figure 4.



**Figure 4. Three Types of Gate State Machine.**

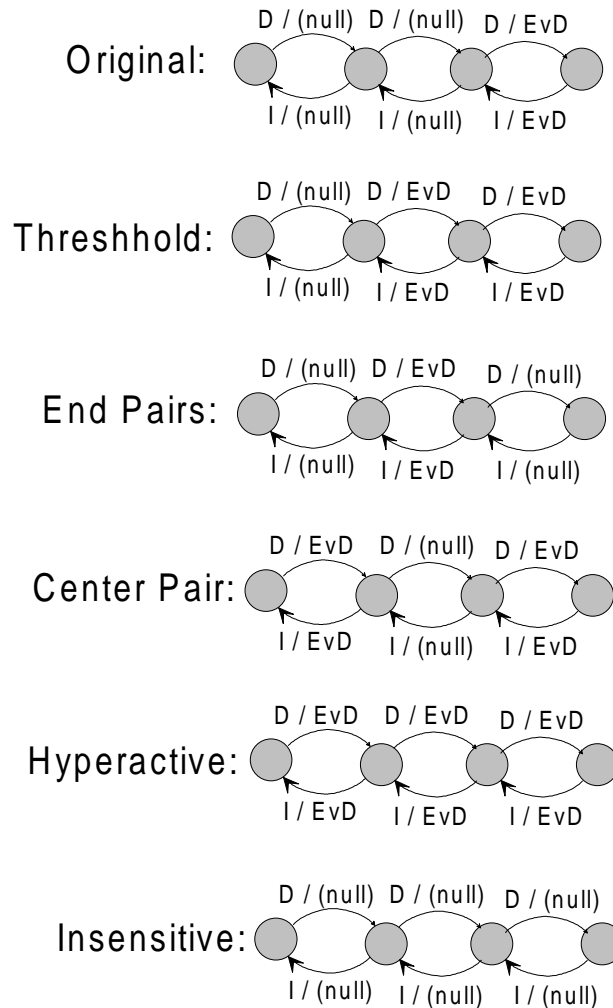
The Hyperactive machine can be used to simulate the XOR and XNOR functions, while the Insensitive machine can be used to simulate constant one and constant zero. The four additional 2-input functions, Not A, Not B, A and B, can be simulated using the Hyperactive machine, as long as the Mealy outputs are removed from one of the Input state machines. These functions can also be simulated by less elaborate mechanisms.

The true power of the linear state-machine becomes more obvious when dealing with functions of more than two inputs. Figure 5 illustrates the state machines used by 3 and 4-input AND gates. (The simplest linear state machine is that illustrated in Figure 1.)



**Figure 5. Three and Four-Input Linear State Machines.**

As with 2-input gates, the input and output state machines can be modified, so the basic 3- and 4-input state-machines can simulate 16 and 32 different functions respectively. It is also possible to alter the gate state machines. Figure 6 gives the variations of the 3-input linear state machine.



**Figure 6. 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  $2$  to the power  $2n+1$  different functions. However, there are  $2$  to the power  $2^n$  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 cubic state machine illustrated in Figure 7. This machine can be used to simulate any 3-input function. The general form of the cubic state machine is capable of simulating any  $n$ -input function but requires  $2^n$  states to do so. The cubic state machine of Figure 7 is simply the cross product of three machines of the type illustrated in Figure 1, 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 8.

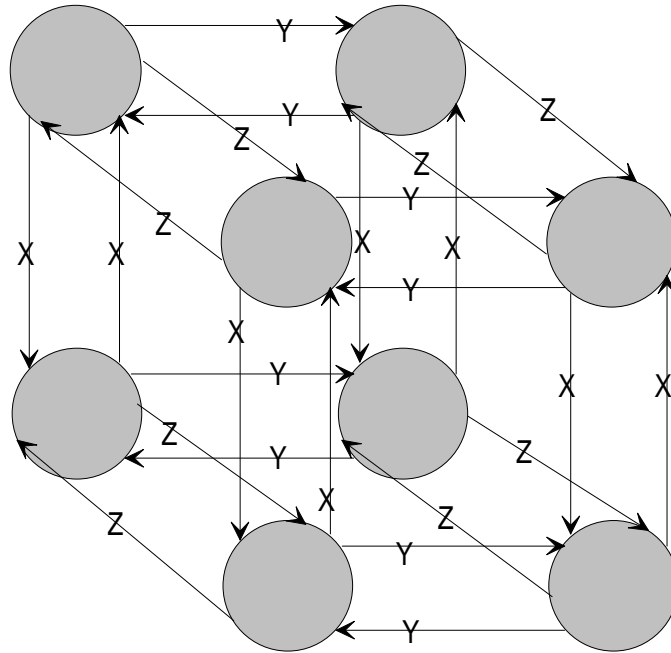


Figure 7. A 3-Input Cubic State Machine.

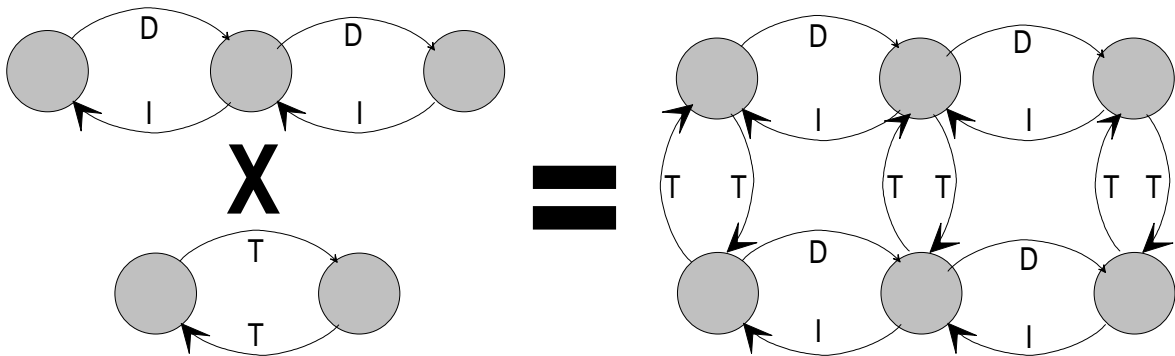
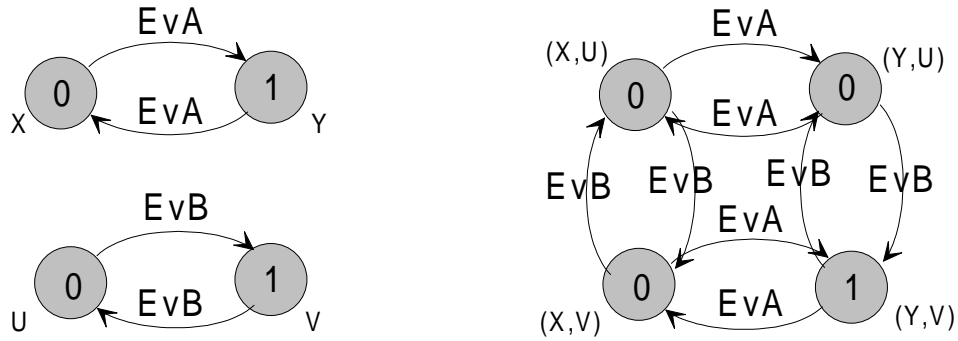


Figure 8. The Cross Product of Two Linear State Machines.

### 3 Systematic Generation of State Machines

All state machines used for inversion-algorithm simulations can be derived from the basic state machine pictured in Figure 1. The first step in creating the gate state machine for a function is make a cross-product machine, then assign Moore outputs based on the function. For example, 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 illustrated in Figure 9.

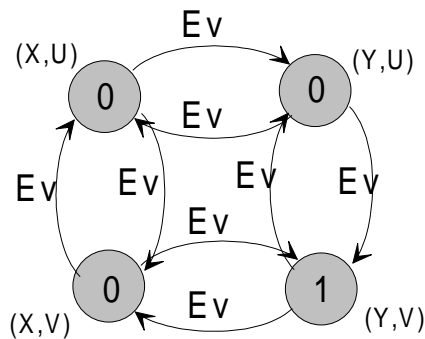


**Figure 9. Developing a Gate State Machine.**

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.

The next step in creating a gate state machine is to identify symmetric inputs. Eventually this should lead to some linearization of the state machine. When two or more inputs of a function are symmetric, the output of the function depends only on the number of ones and zeros on the inputs, not on the precise configuration of ones and zeros. The linear state machine is a straightforward way to keep track of the input count. Even though two inputs might be symmetric, it is still necessary to maintain some information about the individual states of the inputs. A change in two different inputs will generally cause a different result than two consecutive changes in a single input. The Mealy outputs of the input state machines are used for this purpose.

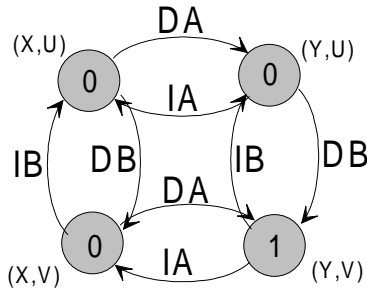
In Figure 9, it is desirable to collapse the symmetric inputs A and B by replacing the events EvA and EvB with a single event Ev. This will yield the non-deterministic state machine illustrated in Figure 10.



**Figure 10. A Naïve Attempt to Combine Symmetric Inputs.**

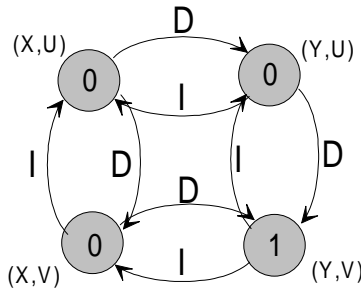
Before the state machine of Figure 10 can be implemented, it is necessary to convert it into a deterministic machine. This will eventually produce a state of the form  $\{(X,V),(Y,V),\dots\}$ . Because the two original states  $(X,V)$  and  $(Y,V)$  have different Moore outputs, the Moore output of the new state will be ambiguous. For some applications, such ambiguous states may not cause a problem. However, for gate state machines, the presence of such a state indicates that the gate does not retain sufficient information to simulate the gate correctly. The use of the Mealy outputs I and D in the input state

machines can resolve the ambiguity for this example. The first step is to recast the state machine of Figure 9 in terms of the I and D inputs, as illustrated in Figure 11.



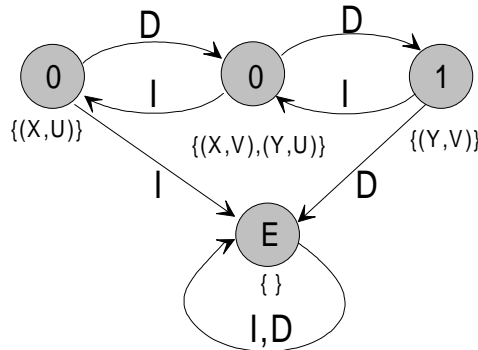
**Figure 11. Using I and D Inputs in a Cross-Product Machine.**

Using the state machine of Figure 11, it is possible to combine the A and B inputs without creating an ambiguous machine. Combining these inputs results in the non-deterministic machine illustrated in Figure 12.



**Figure 12. Non-Ambiguous Combined Inputs.**

Again, the state machine of Figure 12 is non-deterministic and before it can be implemented, it must be converted to a deterministic machine.. The equivalent deterministic state machine is given in Figure 13.

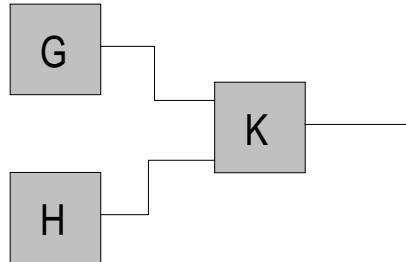


**Figure 13. The Equivalent Deterministic Machine.**

Note that, except for the Error State, the state machine of Figure 13 is precisely the gate state machine illustrated in Figure 2. The reason the Error State was missing from Figure 2, 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.)

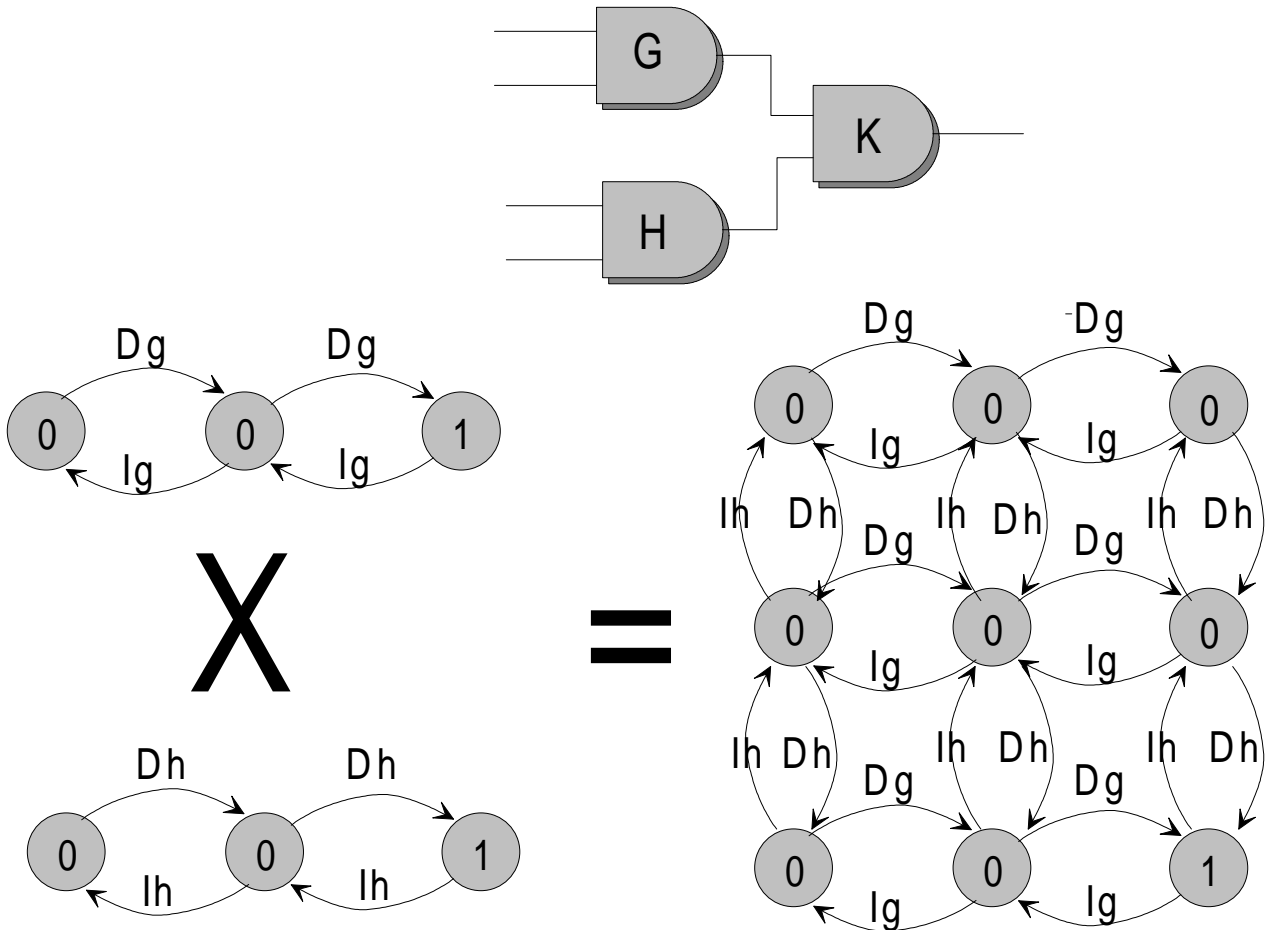
Although a the gate state machine for a function can always be created starting with simple binary machines, it is sometimes easier to treat the function as a collection of

simpler functions, and build the gate state machine. 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.)



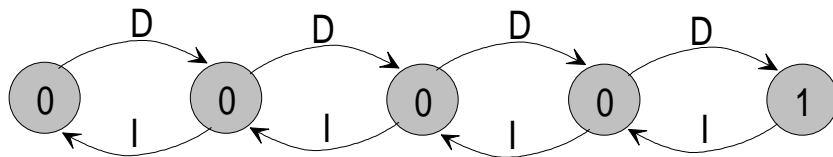
**Figure 14. A Collection of Gates.**

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.



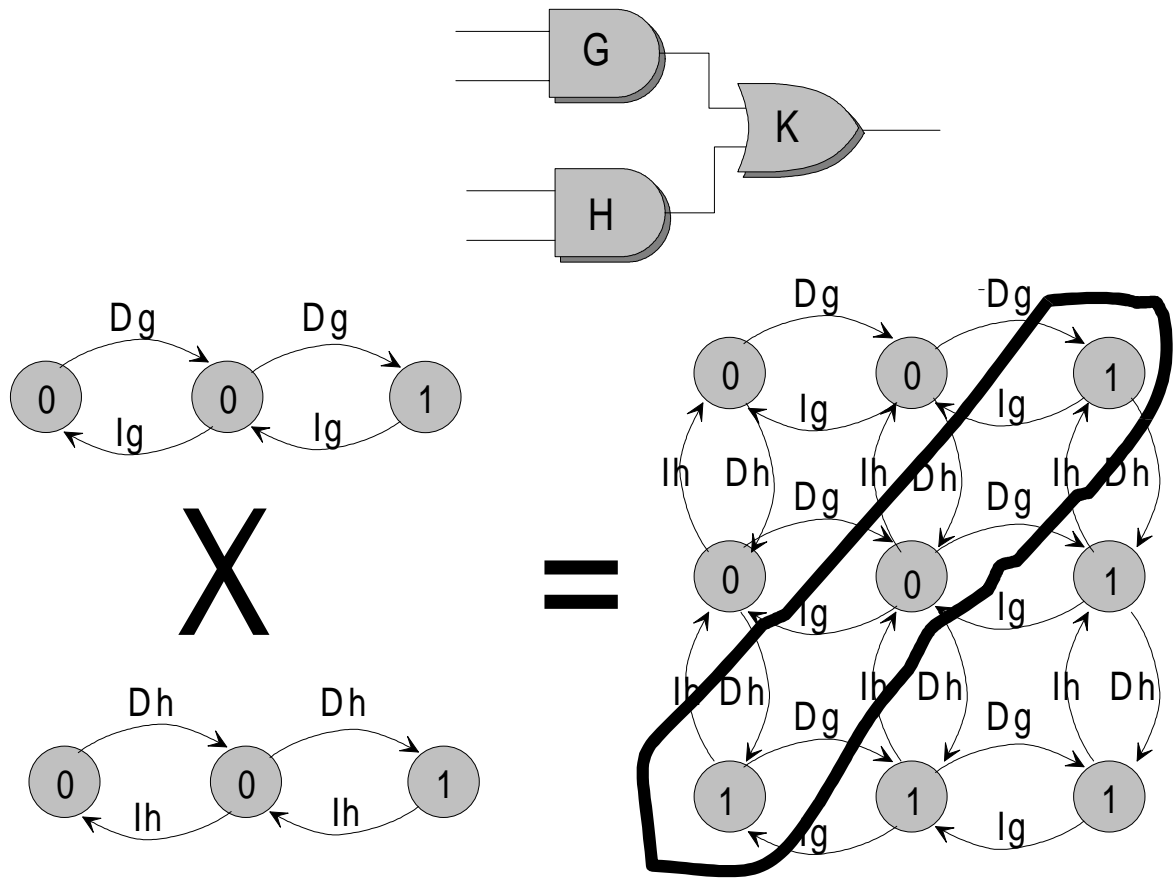
**Figure 15. Using State Machines to Collapse Homogeneous Connections.**

In Figure 15, it is possible to combine symmetric inputs using the rules  $D_h = D_g = D$ , and  $I_h = I_g = I$ . This will yield a non-deterministic state machine which must then be converted into 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.**

In Figure 17, if the inputs of the state machine are combined as before, the resulting non-deterministic machine will be ambiguous. The three circled states will be combined into a single state when the machine is converted to a deterministic machine. Because the Moore outputs of these states are not all the same, the resultant deterministic machine will be ambiguous.

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. As noted above, once the Mealy outputs have been assigned, the Moore outputs can be dropped.

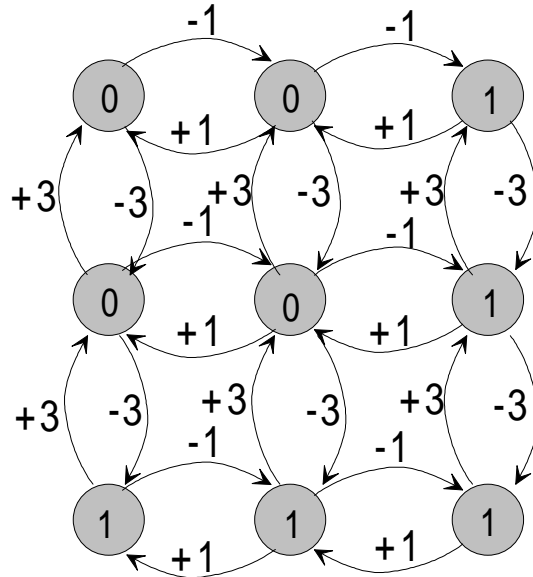
#### 4 Implementation of State Machines

In the Inversion Algorithm, all state machines are implemented by the event processing routines of the various nets. The binary transitions of the nets themselves are handled by the alternation of two different subroutines. The first subroutine produces the I output, while the second produces the D output. Each routine is responsible for scheduling its counterpart. The precise mechanism for doing this is described in [1].

Once scheduled for execution, the primary function of an event processing routine is to handle the state transition and event propagation of a gate state machine. Counting is the most natural way to handle linear state machines, using the I input to increment the count and the D input to decrement the count. The Mealy outputs of the gate state

machine are used to determine when to propagate an event. Non-NULL Mealy outputs cause event propagation, while NULL Mealy outputs do not. Detecting non-NULL Mealy outputs can be done by examining the resultant state after the increment or decrement has been performed. For the standard state machines illustrated in Figure 2 and Figure 5, a test for one after an increment or zero after a decrement will suffice to detect all non-NULL Mealy outputs. For more complex linear state machines, it may be necessary to test for several different values.

The procedure for handling cross-product machines is best understood by considering the cross product of two linear state machines A and B. Assume that the inputs to A are  $I_a$  and  $D_a$ , while the inputs to B are  $I_b$  and  $D_b$ . In the cross product machine, the inputs  $I_a$  and  $D_a$  are handled exactly as they are in the original machine A. In other words, the input  $I_a$  causes the count to be incremented by 1 and the input  $D_a$  causes the count to be decremented by 1. To handle the inputs  $I_b$  and  $D_b$ , it is first necessary to determine the maximum value of the count in machine A. Suppose that this maximum value is  $k$ . The input  $I_b$  will cause the count of the cross-product machine to be incremented by  $k+1$ , while the  $D_b$  input the count will be decremented by  $k+1$ . This principle is illustrated in Figure 18, which shows the state machine of Figure 17 with increment and decrement values added.

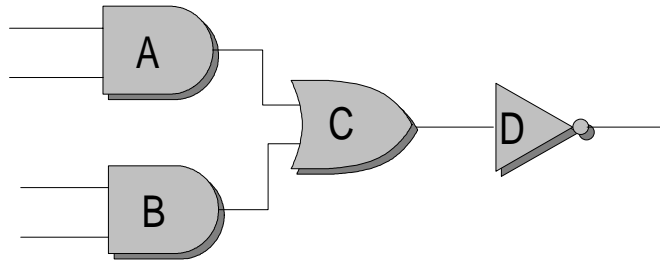


**Figure 18. Cross-Product Machine with Increments and Decrements.**

To fully characterize cross-product state machines, it is necessary to define the concept of a general counting machine. All linear state machines are general counting machines. Furthermore, a cross product of two or more general counting machines is also a general counting machine. The cross product  $M$  of two general counting machines,  $A$  and  $B$  is created by first forming the cross-product state diagram, and then adding values to the transitions. For the transitions due to machine  $A$ , the transition values for  $M$  are identical to the transition values for  $A$ . Let  $k$  be the maximum value of the count for  $A$ . If a transition has value  $q$  in state machine  $B$ , then any transition in  $M$  that is due to machine  $B$  has value  $q \times m$ . Cross products of more than 2 general counting machines can be created by forming cross products two at a time. That is,  $A \times B \times C = (A \times B) \times C$ . The

cubic state machines, such as that illustrated in Figure 7, are also general counting machines. Thus general counting machines are powerful enough to simulate any Boolean function.

The primary difficulty with general counting machines is detecting non-NULL Mealy transitions. One way to simplify the tests for non-NULL Mealy transitions is to use multiplication instead of addition as the state-transition function. Rather than using states of the form 0, 1, 2, 3, ... one uses states of the form  $2^0, 2^1, 2^2, 2^3, \dots$ . Instead of incrementing and decrementing the count by the value  $k$ , the count is multiplied or divided by the value  $2^k$ . 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. Each bit position in the counter represents a different state, making it 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  $s \in S$ .



<p>IA Transitions:            Count = Count &lt;&lt; 1;            if ((Count &amp; 0x24) != 0)            {                Queue Following Event;            }            SP = SP-&gt;Next;            goto *SP-&gt;RTN;</p>	<p>IB Transactions:            Count = Count &lt;&lt; 3;            if ((Count &amp; 0xC0) != 0)            {                Queue Following Event;            }            SP = SP-&gt;Next;            goto *SP-&gt;RTN;</p>
<p>DA Transitions:            Count = Count &gt;&gt; 1;            if ((Count &amp; 0x12) != 0)            {                Queue Following Event;            }            SP = SP-&gt;Next;            goto *SP-&gt;RTN;</p>	<p>DB Transactions:            Count = Count &gt;&gt; 3;            if ((Count &amp; 0x18) != 0)            {                Queue Following Event;            }            SP = SP-&gt;Next;            goto *SP-&gt;RTN;</p>

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

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 the multiply operator. (The inversion of the Moore outputs does not affect the structure of the machine.) This implementation 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 have non-NULL Mealy outputs are the transitions 4->5 and 1->2, while the two  $D_b$  transitions that have non-NULL Mealy outputs are 7->4 and 6->3. The binary masks for these two types of transitions are 000,100,100 and 000,011,000. The  $D_a$  and  $I_b$  transitions are left as an exercise for the interested reader.

## 5 Conclusions

The state-machine analysis presented in this paper can significantly increase the simulation power of the Inversion Algorithm. The techniques described here allow many different types of functions to be simulated as a single gate. The original Inversion Algorithm was limited to a relatively small class of symmetric functions. Complex functions could be handled only as layered simple functions. This paper presents a greatly simplified method for handling heterogeneous connections, and opens the door to additional research on complex state machines and symmetry classes.

The size of a computer word and the choice of state transition function both limit the size of the gate state machines. If the maximum word size is 32 bits, cubic state machines can be constructed for gates with as many as 32 inputs, as long as addition is used as the state transition function. If multiplication is used as the transition function, then the number of inputs is limited to 5. Combining symmetric inputs can significantly expand the capacity of a single word. Using addition for the state transition function, it is possible to handle totally symmetric functions with as many as  $2^{32}$  inputs. (Using multiplication would limit this to 32 inputs.)

Although it is possible to construct large cubic state machines, the complexity of testing for non-NULL Mealy transitions becomes significant as the size of the machine expands. A cubic state machine with  $n$  inputs has  $2^n$  states. On a state transition, it may be necessary to test for as many as  $2^{n-2}$  different states. (Since the direction of the transition is known, this eliminates half the states from consideration. When testing for non-NULL Mealy transitions, there are two sets of states, those that imply non-NULL Mealy transitions and those that do not. It is sufficient to test for the smallest set.)

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 Inversion

Algorithm. Because it is possible to remove all NOT gates from an Inversion Algorithm simulation, 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 as far as the Inversion Algorithm is concerned, *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 any existing Inversion Algorithm implementation, and the problems described here will serve as the basis of much future research in improving the performance of the Inversion Algorithm.

## 6 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.