VLSI DESIGN
1995, Vol. 3, Nos. 3-4, pp. 249-265
Reprints available directly from the publisher
Photocopying permitted by license only

FSM Decomposition and Functional Verification of FSM Networks[1]

ZAFAR HASAN and MACIEJ J. CIESIELSKI
Department of Electrical & Computer Engineering, University of Massachusetts, Amherst, MA 01003

Here we present a new method for the decomposition of a Finite State Machine (FSM) into a network of interacting FSMs and a framework for the functional verification of the FSM network at different levels of abstraction. The problem of decomposition is solved by output partitioning and state space decomposition using a multiway graph partitioning technique. The number of submachines is determined dynamically during the partitioning process. The verification algorithm can be used to verify (a) the result of FSM decomposition on a behavioral level, (b) the encoded FSM network, and (c) the FSM network after logic optimization. Our verification technique is based on an efficient enumeration-simulation method which involves traversal of the state transition graph of the prototype machine and simulation of the decomposed machine network. Both the decomposition and verification/simulation algorithms have been implemented as part of an interactive FSM synthesis system and tested on a set of benchmark examples.

Key Words: *Finite State Machine, FSM Synthesis, FSM Decomposition, Performance optimization, Multiway partitioning, Verification, Simulation, Enumeration-simulation, CAD Tools*

1. INTRODUCTION

Sequential circuits play a major role in the control part of digital systems and efficient computer-aided design tools are essential for their design. Of particular interest are systematic methods for the synthesis of finite state machines (FSM) by means of functional and physical decomposition. Implementing a finite state machine as a network of interacting submachines can be advantageous as it improves the performance of FSM controllers; this, in turn, may significantly affect the system clock. Also, in most cases, the size of the machine can be significantly reduced. Decomposition of FSMs is particularly useful when the Field Programmable Gate Array (FPGA) or Programmable Logic Device (PLD) technologies are used for their implementation. The FPGA and PLD logic is realized by means of interacting logic blocks, with restrictions on the number of I/O lines per block and sometimes on the number of product terms per block [15]. In many cases it is desirable, for reasons of clock-skew minimization or simplifying the layout, to distribute the control logic for a data path in such a manner that portions of the data path and control that interact closely are placed next to each other. FSM decomposition can be used for this purpose as well. Complex systems obtained from high-level specifications using VHDL may also be implemented as networks of FSMs. Therefore there is a definite need for a decomposition and verification system to help the designer in the synthesis of complex digital controllers.

In this paper we address the problem of decomposition and verification of sequential machines. Several algorithms have been proposed to decompose an FSM into two interacting submachines [17], [10], [3], [14], but no significant results have been achieved in the field of multi-machine decomposition. Here we present a technique to decompose an FSM into an arbitrary number of submachines. Our decomposition approach aims at optimizing the performance of the resulting implementation. This is in contrast with other methods, such as [3], where the emphasis is on reduction of circuit area only. Our approach is unique in that the number of submachines is not predetermined, but is determined dynamically, depending on the characteristics of the original machine.

[1]This work was supported in part by the NSF under grant number MIP-9013013 and MIP-9208267.

Furthermore it provides a systematic way to distribute outputs among the submachines, prior to determining the internal states of the submachines. In previous work, outputs have been assigned to machines either in an arbitrary fashion or after the internal states of the machine have been already determined. Multiway partitioning has been used here to optimally determine the internal states of the submachines. Previous approaches to FSM decomposition have used the number of states and the number of edges in the resulting submachines as their cost function (e.g. [17], [10]). Given that the logic implementation of an FSM is derived from its state transition graph (STG) specification, followed by state assignment and intensive logic optimization, this cost function does not reflect the true complexity of the eventual logic-level implementation and is often far from accurate. Our technique uses a more accurate estimate based on symbolic minimization of the FSM.

The process of decomposition, encoding, and synthesis of an FSM can be very complex and time consuming. There are many sources of errors that can produce an incorrectly functioning circuit. Designers often use various verification techniques at different stages of the synthesis process to obtain an error-free system. Errors could be introduced in the specification or implementation. *Design verification* is the process of determining whether the original specification is correct. Once the specification is verified, an implementation of the circuit is derived. At this level an incorrectly functioning circuit could be a result of either a human error or an error introduced by an automatic design tool. *Implementation verification* is the process of determining whether the designed circuit meets the original specification. *Logic verification* is the process of verifying the equivalence of two logic-level circuits, usually the optimized and the optimized one [12]. Reliable verification tools are necessary to ensure the correctness of the final design.

Given a specification of a sequential machine and its decomposed version the goal of the functional verification is to verify the correctness of the design with respect to the original specification. To the best of the authors knowledge very little work has been done on the functional verification of decomposed machines with respect to the specified prototype at the behavioral level, with the exception of the work of Jóźwiak [20]. There are few efficient tools available today which can verify the system at various stages of the synthesis process. Most of the verification methods reviewed in Section 3 are aimed only at logic verification, i.e., the verification of two implementations of the same machine.

Our approach to verification is a modified form of the enumeration-simulation approach to verification [11], allowing it to be used for networks of FSMs at the behavioral as well as at the logic level. The method is general enough to be used at different stages of the design process. The need for an algorithm to verify two circuits at differing levels of abstraction was shown in [24], [11]. Both the decomposition algorithm and the verification/simulation algorithm have been implemented as part of a larger interactive FSM synthesis system being developed at the university of Massachusetts at Amherst. The decomposition program can be used to decompose a given FSM into a network of interacting submachines. The verification program can be used for design verification after decomposition of the prototype machine, or for simulating the design. Implementation verification can be performed after the encoding of the submachines. It can also be used for logic verification of the optimized submachines. The tool has an additional decomposition generation option to help in the design of FSM networks. Although we have demonstrated the use of our method for the verification of FSM networks obtained from the decomposition of a single FSM, it can readily be used to verify FSM networks obtained from high level specifications.

The rest of the paper is organized as follows. Section 2 introduces basic definitions and notation. Section 3 briefly reviews some of the earlier work on decomposition and verification. Section 4 contains the details of the decomposition method and presents some experimental results. Section 5 describes our verification algorithm in detail, with section 5.2 describing the enumeration-simulation technique. The other subsections describe the application of the verification algorithm to the various cases covered in this paper and report results of the verification technique. The last section summarizes our work on decomposition and verification.

2. PRELIMINARIES

A finite state machine M can be described by a five-tuple $M = (S, I, O, \delta, \lambda)$, where S is a set of state symbols, I is a set of primary inputs, O is a set of primary outputs, $\delta : I \times S \rightarrow S$ is the next state function, and $\lambda : I \times S \rightarrow O$ is the output function (Mealy machine). An FSM can be represented by its *State Transition Graph (STG)* or equivalently, by its *State Transition Table (STT)*. Figure 1(a) shows a general sequential circuit.

Definition 2.1 *A* **partition** π *on a set of states S is a collection of disjoint subsets of S, called* π *blocks, whose set union is S. A partition* π *on the set of states S of a machine M is said to be a* **closed partition** *if and only if for any two states s and t which are in the same block of* π*, and for any input* $i \in I$*, the next states* $\delta(s,i)$ *and* $\delta(t,i)$

are in a common block of π [22]. *A partition is a* **general partition** *if it is not closed.*

We refer to the original machine as the *prototype machine,* and to the individual machines that make up the overall realization as *submachines.* The machine obtained as a result of the decomposition is called the *decomposed machine* and is implemented as a network of submachines. The general structure of an FSM network obtained from a general partition is shown in Figure 1(b).

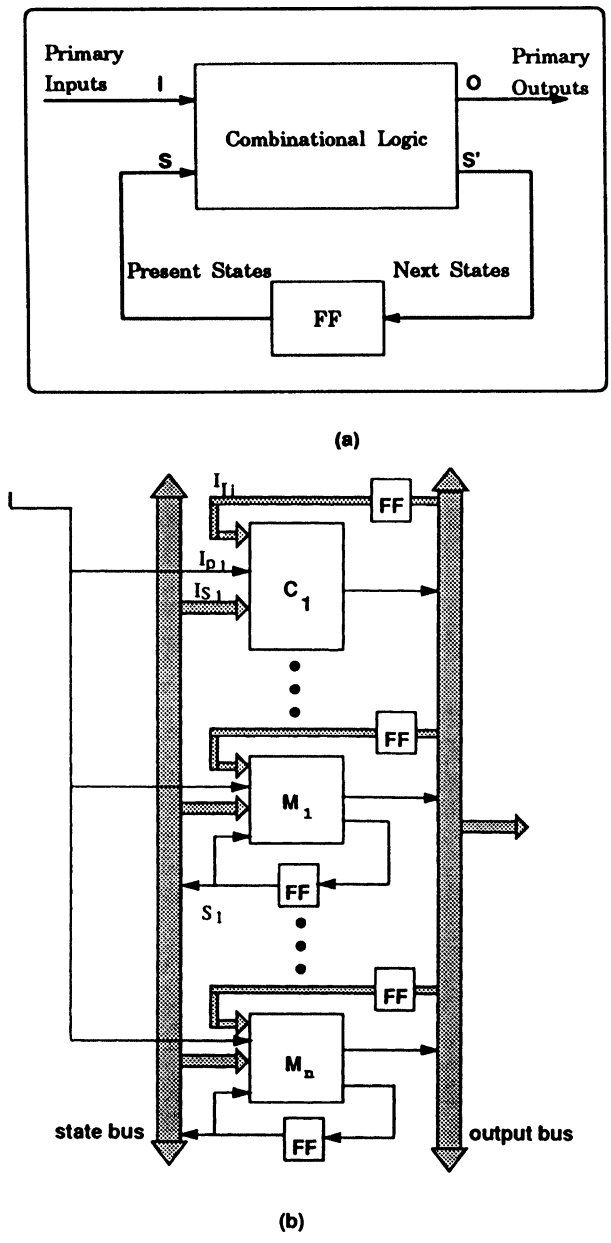


FIGURE 1 (a) General sequential circuit (b) General decomposition topology.

Definition 2.2 *The product of two partitions,* π_P *has a block in* π_P *for every pair of blocks in the two partitions, where the elements of the block are the intersection of the elements in the block pair. If the intersection is a null set then the block is not present in* π_P.

For example, the product of $\pi_1 = \{A,B: C,D\}$ and $\pi_2 = \{A,C : B,D\}$ is $\pi_P = \pi_1 \cdot \pi_2 = \{A : B : C : D\}$. Here the first block of π_P was obtained by taking the intersection of $\{A,B\}$ and $\{A,C\}$. The other blocks are obtained in a similar fashion.

Let $\pi(0)$, called the *zero partition,* denote a partition with $|S|$ blocks, such that each block contains exactly one state. It can be shown that a machine M can be decomposed into a set of n interacting machines that perform the same function as M if and only if there exists a set of nontrivial partitions $\pi_1, \pi_2, \ldots, \pi_n$ such that

$$\pi_1, \pi_2 \ldots \pi_n = \pi(0)$$

Definition 2.3 *A set of partitions* $\pi_1, \pi_2, \ldots, \pi_n$ *whose product is equal to* $\pi(0)$ *defines a* **legal decomposition** *of machine M.*

A legal decomposition satisfies the behavior of the prototype machine. In the decomposition, each component M_i is associated with one partition π_i, and its states represent blocks in π_i. Thus, the number of states in M_i is equal to the number of blocks in the partition. The verification problem can be formulated as a decision problem where, given two descriptions of a circuit, the question is whether the two descriptions represent the same functionality. In order to show that two such circuits are not equivalent, it is necessary to find a primary input sequence which, when applied to the two machines, results in the machines asserting different output sequences. If no such sequence exists, the machines are said to be *equivalent.* Before trying to determine the equivalence, a correspondence between at least one state in each machine is required. Therefore each machine is assumed to have a *reset state.* The problem of verification of sequential machines is thereby reduced to the problem of determining the equivalence of the reset states of the two machines [22]. What this implies is the verification of all possible transitions of the machine starting from the reset state which effectively verifies the equivalence of the two machines.

Definition 2.4 *For a pair of states* (s_i, s_j), *if there exists a differentiating sequence of length k, the states* s_i *and* s_j *are said to be* **k-differentiable.** *States that are not i-differentiable for any* $i \leq k$ *are said to be it* **k-equivalent.** *If a finite differentiating sequence does not exist for a state pair, the states in the pair are said to be* **equivalent.**

3. PREVIOUS WORK

The decomposition of sequential machines was first treated in a formal way by Hartmanis and Stearns [17]. They proposed two types of decomposition, parallel and cascade, based on the topology of the decomposed machine. In *parallel decomposition* the submachines are supplied with the same input sequence, but operate independently. There is no interaction or exchange of information between the submachines. Parallel decomposition has limited use in the design of modern finite state machines. Practical designs do not usually have good parallel decompositions. Another type of decomposition is the *cascade* or *serial decomposition*. Here also each submachine is driven by the same input sequence, but the submachines do not operate independently. One submachine is supplied, by means of auxiliary inputs, with information about the current internal state of the other. This information influences the state transitions of a submachine and enables it to generate the appropriate output sequence. The possibility of passing state information between the submachines makes cascade decomposition more powerful than parallel decomposition. However the transmission of state information is serial. A submachine requires state information of its own states and about the states of its predecessors. It feeds its own state information to its successor machines.

In another form of decomposition, presented by Devadas and Newton [10], both components of the decomposed machine interact with each other. This form of decomposition involves identifying *subroutines* or *factors* in the original machine, extracting these factors and representing them as a separate *factoring machine*. The occurrence of these factors become calls to the factoring machine from the factored machine. This method does not have definite objective function to optimize and does not give a clear indication about the quality of the decomposition.

Ashar *et al.* [3] presented another decomposition method where the topology of the decomposed machine is a general decomposition topology similar to that shown in Figure 1. Optimum and heuristic algorithms for two-way general decomposition of FSMs are given, such that the total number of product terms in the one-hot coded and symbolically minimized submachines is minimal. They use the number of product terms in the decomposed machine as the cost function. Moreover, the objective of the decomposition is to minimize the area of the final implementation. The problem of optimum two-way FSM decomposition is formulated as one of symbolic output partitioning. A procedure of constrained prime implicant generation and covering is described for optimum FSM decomposition under a specified cost function.

Recently a multiway decomposition algorithm has been proposed by Jóźwiak *et al.* [19] [18]. The topological structure of their decomposed machine is different from the one proposed here. An additional function is used to generate the primary output from the outputs of individual submachines. This provides greater flexibility in partitioning into submachines but requires an extra combination logic block to realize the final outputs. It is not clear which scheme is better for any given situation. This is a constraint based decomposition scheme with the aim to generate submachines which can fit in predefined logic blocks. The adopted cost function is the number of blocks used and the number of communication lines between the blocks.

Most of the above mentioned decomposition methods basically aim at two-way partitioning. There is no significant work done on multiway partitioning of sequential machines except [18], where multiway partitioning is applied using a recursive scheme. In principle, the general decomposition method described by Ashar *et al.* [3] can also be extended to multiway partitioning, but the method is not clearly indicated. Furthermore, none of the published work aims at explicitly minimizing the delay of the decomposed machine.

The problem of verification of single FSMs has been under investigation for a long time. Some of the algorithms are based on the verification of the product of the two machines [13], others on graph traversal, enumeration-simulation [11], and symbolic STG traversal [8]. Most of the reported work on FSM verification deal with equivalence checking and few of these techniques have been widely used in practice due to their low efficiency.

The basic principle of enumeration-simulation approach, first presented in [11], is that all state transitions of one machine are enumerated and simultaneously simulated on the other machine. During the enumeration every path in the STG and all valid states have to be visited. A depth-first enumeration approach is used whereby only one path in the STG has to be stored at any point of time, making the approach memory efficient. Our method is based on a modification of this approach to make it even more efficient. A similar approach was presented in [16], however, it considers the verification of a single machine after encoding. We extend this idea to verification of decomposed machines at the symbolic level.

Some very efficient *symbolic STG traversal* algorithms have been developed that can be used to traverse the STG of a machine and verify whether a certain property is true for all the valid states of the machine [7] [8]. In these approaches, a breadth-first technique is used and the input as well as the state space is implicitly enumerated. The algorithms are best implemented using Binary Decision Diagrams (BDD) [6]. These methods work best

for circuits that have certain regularity in their STG structure. Also, for certain machines, the transition relations become very complex causing each iteration to take a long time. Furthermore, the BDDs can be constructed only after the machines have been encoded and the method cannot be used for verification at the symbolic level; as such, this method is not very useful for our framework.

There exist some other algorithms that can verify single FSMs but require huge amount of memory for storage and can be used only for small machines, [5] [24]. Most of the above mentioned algorithms verify two implementations of a single FSM at the logic level. They do not address the problem of verifying a network of FSMs at the symbolic level. Improved performance requirements will force the larger FSMs to be decomposed and implemented as a network of FSMs. This will require efficient tools to verify these networks at various stages of the synthesis process. Recently, a verification method for decomposed FSMs was presented by Jóźwiak [20]. Verification is done by the reverse mapping of the decomposition steps to generate the original specification. Thus it becomes necessary for the method to know the decomposition steps. Our framework can verify a decomposed FSM at various levels of abstraction with its original prototype specification given at the symbolic level even without having the decomposition information.

4. DECOMPOSITION

The problem of FSM decomposition can be formulated as that of assigning output bits to the submachines and obtaining a legal decomposition on the set of states for each submachine. The original states are partitioned in such a way that each block of a partition represents one internal state of the submachine and is assigned a distinct code in that submachine. To attain a legal decomposition and differentiate all states of the prototype machine, the product of all such defined partitions must be $\pi(0)$.

4.1 The Architecture

The architecture of the decomposed finite state machine is as shown in Figure 1(b). We refer to this topology as the *general decomposition* topology since it can represent an arbitrary decomposition. Parallel and cascade decompositions are special cases of the general decomposition. In this topology each submachine M_i has three sets of symbolic inputs.

1. A set of **primary inputs** I_{p_i}; these inputs can be either symbolic or binary, depending on the initial specification.
2. A set of it **state inputs** I_{S_i}, derived from the states of other submachines; for a machine corresponding to a closed partition, $I_{S_i} = \emptyset$.
3. A set of it **internal states**, S_i.

The primary inputs are made available to every submachine. The states of one machine are made available to the others whenever necessary. The outputs are generated directly by individual submachines instead of using another combinational block to generate the outputs (recall [19]). Note that state information is being shared by the submachines and that, in general, the submachines are dependent on each other; this clearly affects the encoding of the submachines. A global encoding technique described in [25], that considers the interaction between the submachines and minimizes the communication complexity in the FSM network, can be used to efficiently encode the machines.

Example 1: Consider a prototype machine M with 4 primary outputs, Q_1, Q_2, Q_3, Q_4, and 4 states, $S = (A : B : C : D)$. Suppose this machine is decomposed into 2 submachines, M_1 and M_2, as shown in Figure 2.

In machine M_1, original states A and C will get the same code; similarly states B and D will get the same code. In machine M_2, states A and B will get the same code, etc. Each submachine will now require the state information from the other submachine in addition to the primary input, to distinguish the original states. Consider an entry in the STT of the original machine M.

Prototype Machine

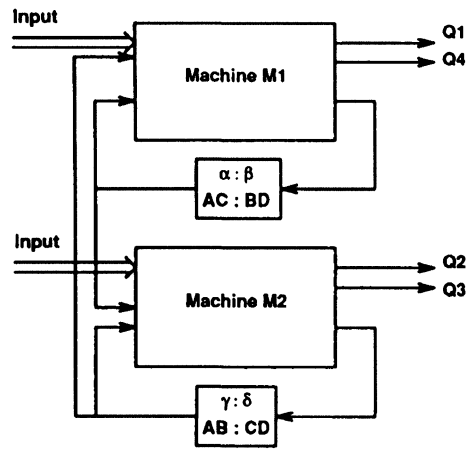
I_p	S	S'	Q
11	B	C	1011

Since state B is assigned symbolic state β in M_1 and state γ in M_2, the same entry in submarine M_1 is represented as followsSubmachine *M1*

I_{p1}	I_{S_2}	S_1	S'_1	Q_1	Q_4
11	γ	β	α	11	

4.2 The Decomposition Algorithm

The primary objective of our approach is to reduce the complexity of each submachine (which helps in improv-



Prototype:

States: A, B, C, D
Outputs: Q_1, Q_2, Q_3, Q_4

Decomposed:

M_1 : *States* $(AC : BD) = (\alpha, \beta)$
Output $= Q_1, Q_4$
M_2 : *States* $(AB : CD) = (\gamma, \delta)$
Output $= Q_2, Q_3$

FIGURE 2 Simple machine decomposition, example 1.

ing the performance of the overall system), while attempting to keep the total number of submachines small. The decomposition algorithm works in two main steps: (1) partitioning of the outputs and (2) partitioning of the states. The decomposition algorithm presented here is different from most of the existing methods in this regard. Most of the earlier FSM decomposition techniques deal only with the state partitioning problem and randomly distribute the outputs among the submachines once the states have been partitioned. The algorithm presented here does an explicit assignment of outputs to the submachines so as to achieve the best possible results. The adopted cost measure is the estimated performance and area, based on a two-level implementation strategy. We use the number of product terms as the measure of circuit complexity as this is the most accurate estimate possible; this measure is readily available using the method of symbolic minimization.

The first step of our decomposition procedure is to partition the output bits among the submachines. The output partition automatically determines the number of submachines in the final decomposition. This step is followed by the partitioning of the original states of the prototype machine to determine the internal states of each submachine; this is done for each of the submachines separately. Output partitioning is chosen as the first step because the assignment of outputs to a submachine significantly affects the structure of that submachine and the subsequent state partitioning.

4.2.1 Partitioning of the outputs—The cost of a partition of output bits can be approximated as a combination of the area cost and the performance cost, with suitable weights depending on the desired objective function. Given the output bits assigned to a submachine, the cost cannot be exactly determined without knowing the partitioning of the states or the final implementation of the submachine. However it can be estimated by finding the number of product terms assuming two-level implementation and one-hot encoding of the states. The area cost of a submachine is estimated using the following equation

$$Area = (2 \cdot I + O) \cdot P \tag{1}$$

where I is the total number of inputs (primary inputs, external state inputs, and internal state inputs), O is the number of outputs, and P is the number of product terms in the submachine. The performance of a circuit is determined by the delay along the longest path. Based on the model presented in [14], the delay for a PLA implementation is estimated as

$$Delay = max_{(i,j)}(f_{I_i} + f_{A_j}) \tag{2}$$

where f_{I_i} is the fanout for the i^{th} input line I_i, and f_{A_j} is the fanout for the line corresponding to j^{th} product term. This can be rewritten as

$$Delay = max_{(i,j)}(K_{I_i} P + K_{A_j} O) \tag{3}$$

where P is the number of product terms and K_{I_i} is the fraction of product terms used by the i^{th} input line. O is the number of outputs and K_{A_j} is the fraction of outputs connected to the j^{th} product term; K_I, $K_A \in [0,1]$. For estimation purpose it is assumed that $K_I = K_A = 0.51$ as determined experimentally in our earlier work [14] (the experimental values justify the assumption of $K_I = K_A$).

The output bits of the original machine are grouped together on the basis of maximum saving in area and performance. Each group of outputs will be assigned to one submachine in the final implementation. The algo-

rithm uses ideas from the simulated annealing technique. Starting from a random partition, the cost is determined by an estimated number of product terms for each group in the partition (to be illustrated by the example below). Taking one output at a time, the algorithm finds the best new group to which to assign that output. The gain for each of the moves is calculated, and the move with the greatest gain is chosen. The algorithm randomly chooses one output at a time and attempts to relocate it so as to minimize the cost. Occasionally, random moves are made to avoid a trap in a local minimum. There are several passes; in each pass each output is moved only once. When an attempt has been made to relocate all the outputs, a new pass begins where each output is moved again. The following example illustrates the method for calculating the cost of output partitioning.

Example 2: Consider a partial STT of a prototype machine with 4 output bits as shown in Figure 3. For simplicity we will consider product terms as the cost measure in this example. We can see that implementing this machine as a single FSM requires 5 product terms. Suppose that the random partitioning step generates the following output distribution I: $\{Q_1, Q_3 : Q_2, Q_4\}$ The total cost of this decomposition (the sum for the two submachines) is 9 product terms.

Suppose we were trying to move output Q_2 after the first random partition. There are three possible choices: we can move Q_2 to group $\{Q_1, Q_3\}$, we can move it to a new group of its own, or leave it in the existing group. The move to group $\{Q_1, Q_3\}$ has the maximum gain, so we put it in that group. Similarly, output Q_3 can be moved to the group with output Q_4 or to a new group. Moving Q_3 to the group with Q_4 has the maximum gain, see Figure 4. Hence that move is made resulting in distribution II: $\{Q_1, Q_2 : Q_3, Q_4\}$ with 4 product terms.

4.2.2 Partitioning of the states—The partitioning of outputs uniquely determines the decomposition of the STT of the prototype machine into a set of STTs, each representing one submachine. Sometimes an extra submachine, with no outputs, may be necessary to differentiate all states of the prototype machine and attain a legal decomposition. However, at this point the internal states of the submachines are not yet defined; the individual state tables still refer to the states of the prototype machine. The goal of state partitioning is to create the internal states for each submachine. This is accomplished by grouping the original states of the prototype machine into *blocks* of states, each block forming an internal state of the submachine. This is done for each submachine separately.

The partitioning of the states is solved using a weighted graph $G_{M_i}(V, E_{M_i})$ for each submachine. Each node in V represents a state of the prototype machine. The weighted edges E_{M_i} represent the desired adjacency requirement among the states. The weight assigned to each edge can be either positive or negative. A positive

Prototype Machine M

Input	PS	NS	$Q_1Q_2Q_3Q_4$
I_1	A	B	1100
I_2	A	C	1100
I_2	B	C	1101
I_3	A	B	1111
I_3	B	B	0011

Output distribution I: $\{Q_1, Q_3 : Q_2, Q_4\}$ (after random partitioning)

M_1

Input	PS	NS	Q_1Q_3
I_1	A	B	10
I_2	(A, B)	C	10
I_3	A	B	11
I_3	B	B	01

M_2

Input	PS	NS	Q_2Q_4
I_1	A	B	10
I_2	A	C	10
I_2	B	C	11
I_3	A	B	11
I_3	A	B	01

Cost: 4 + 5 = 9 product terms

Output distribution II: $\{Q_1, Q_2 : Q_3, Q_4\}$

M_1

Input	PS	NS	Q_1Q_2
(I_1, I_3)	A	B	11
I_2	(A, B)	C	11

M_2

Input	PS	NS	Q_3Q_4
I_2	B	C	01
I_3	(A, B)	B	11

Cost: 2 + 2 = 4 product terms

FIGURE 3 The partitioning of outputs, Example 2.

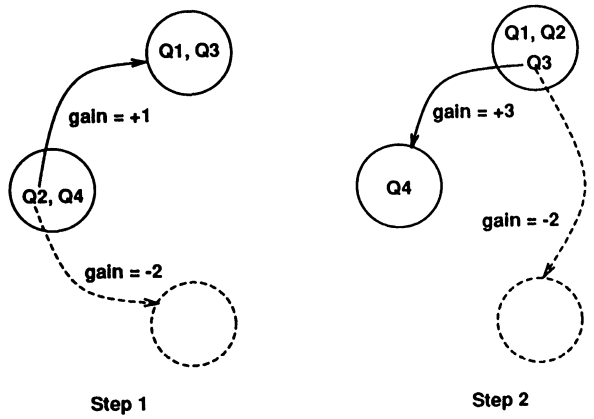


FIGURE 4 Calculation of gain, example 2.

In	PS	NS	Out
I_1	A	B	101
I_1	C	B	101
I_2	D	B	001
I_2	D	D	001
I_1	B	C	111
I_2	E	A	010
I_4	A	E	100
I_4	C	E	100
I_4	E	C	101

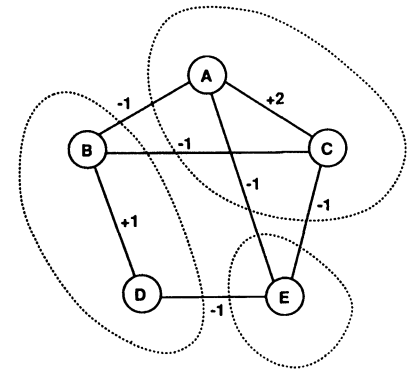


FIGURE 5 Multiway graph partitioning (a) state transportation table (b) weighted graph with possible cuts shown.

weight reflects the extent to which the number of product terms would be reduced, if the two states were given the same code in that submachine. If assignment of separate codes to the states is beneficial, then the edge between the states is given a negative weight. Thus we have *attractive* (positive) and *repulsive* (negative) edges. Multiway partitioning of the graph so constructed is then carried out. The cost of the partition is measured as the sum of the weights of the multiway cut. The idea of having repulsive edges is specially important in the partitioning of the states. The negative weights of these edges indicate that they are the preferred edges for the cut in the multiway graph partitioning.

Obviously the partition of states for a given submachine is affected by the partitioning of states in other submachines. It is impossible to find out how a grouping of states in one submachine affects the overall cost by looking at that submachine in isolation. Therefore, when constructing graph G_{M_i} for a submachine, the algorithm uses information about previously determined state partitions in other submachines by assigning the edge weights accordingly.

The construction of graph G_{M_i} can be briefly described as follows (see Figure 5). A positive weight is assigned to an edge based on the following two rules: (1) An edge with weight $w(e) = 1$ is added between the nodes corresponding to the present states that, for the same input, have the same next state (states A and C in Figure 5). If the edge already exists its weight is incremented by 1. (2) An edge with weight 1 is also added between the next state nodes that are derived from the same present state (states B and D in Figure 5).

A negative weight is added to the edge between nodes representing present states for which, for the same input, the output is different (states A and B). This is needed to distinguish these states in order to generate the output. Although any pair of states can be distinguished with the knowledge of the internal states of other submachines, it is advantageous that these states be assigned different codes in this submachine as this reduces the number of communication bits between the submachines. However, if in some previously designed submachine, these two states have already been differentiated, the weight should be less negative since one does not want to repeat the logic and eventually end up with too many submachines.

The weights assigned to the edges of the state partition graph G_{M_i} represent both the *communication factor* of the machine and the desired *adjacency relations* between the prospective state codes. The rules described above to assign positive weights are aimed at the minimization of the combinational logic component of the FSM. These rules were employed in the past to create a *code adjacency graph* and used to find the state assignment which minimizes the logic complexity of the machine [1], [23]. To obtain the state assignment the adjacency graph was embedded on a minimum-dimension boolean cube so as to minimize the weighted distance on the cube.

The algorithm for multiway graph partitioning uses ideas from the Kernighan and Lin [21] min-cut algorithm for two-way graph partitioning. Starting from a random partition, the algorithm makes iterative improvements to reduce the cost function. Occasionally, the program makes a random move to avoid getting trapped at a local minimum. There are several passes of the algorithm. In each pass the following steps are taken.

- The gains of all possible moves of each node to all other groups are calculated.
- A move is selected either in a greedy way (by picking the move with highest gain, possibly negative) or in a random way to help the program get out of a local minimum.
- The gains are recomputed efficiently by finding out

the incremental gains. The previous step is repeated until all moves are marked rejected. The moves already taken are marked as rejected so that one does not carry out the move again in the same pass.

Multiway partitioning of the graph constructed in this way gives a grouping of states with minimum cost. A penalty is added for the maximum size of the group and the number of nonempty groups. This is to keep the number of internal states small since that too affects the performance and area.

Figure 5 shows the STT of a submachine and the corresponding graph G_{M_i} generated using the rules described above. An optimum partition of this graph is $(AC : BD : E)$, so the submachine will have 3 internal states.

4.3 Decomposition Results

The decomposition method described in this section has been implemented as a C program which is part of a larger interactive FSM synthesis system [26]. We have tested the decomposition program on several examples from the MCNC benchmark suite and some industrial examples. The current version of the program puts greater emphasis on improving performance than area. The results for two-level implementation are shown in Table I where S represents the number of states, O the number of outputs, and P the number of product terms. We used the two-level implementation for our initial estimation. The values for area and delay were calculated using equations (1) and (3) derived in Section 4.2.1, and the table uses the corresponding units.

Some of the decomposed machines were then synthesized as multi-level circuits and mapped to a standard cell library using multi-level synthesis program MISII [4]. The results are shown in Table II. The units for area and delay are different than those in the previous table, so the reported results cannot be directly compared (notice, however, the correlation between the results). The delays reported in Table II are calculated as the sum of the gate delays on the longest path in the circuit using the *delay units* of MISII. Similarly the area of each circuit is calculated as the sum of the areas of the gates using the *grid units* (an approximation of actual area). The delay and area values for each gate are taken from the gate library. Technique described in [25] were used to carry out the encoding of states in the communicating machines.

The results from the table indicate that the delay of the decomposed machine is always reduced and in some cases the area is reduced as well. These results have been

TABLE I
Experimental Results of Two-Level Implementation

Example	Prototype Machine					Decomposed Machine				
	S	O	P	Delay	Area	S	O	P	Delay	Area
bbara	10	2	25	**12**	550	2/2/3	1/1/0	9/9/9	**8**	495
cse	16	7	45	**38**	1485	2/2/4	3/4/0	17/28/27	**25**	1873
dk16	27	3	72	**56**	2880	4/8	2/1	28/47	**26**	3000
dk27	7	2	8	**7**	104	2/2/2	1/1/0	4/4/3	**4**	110
dk512	15	3	19	**7**	323	2/2/2/2	1/1/1/0	5/5/7/6	**5**	276
s1	20	6	83	**47**	3071	2/2/8	3/3/0	29/29/44	**26**	3060
donfile	24	1	48	**25**	960	4/8	1/0	31/25	**14**	952
ex3	10	2	18	**13**	324	2/6	2/0	6/13	**11**	298
pma	24	8	48	**27**	1872	2/2/2/4	1/3/4/0	12/17/14/14	**11**	1686
beecount	7	4	12	**13**	228	2/2/2	2/2/0	7/6/3	**7**	237

TABLE II
Experimental Results of Multi-Level Implementation

Example	Prototype Machine				Decomposed Machine			
	S	O	Area	Delay	S	O	Area	Delay
bbara	10	2	97	30.30	2/2/3	1/1/0	106	14.00
cse	16	7	249	69.10	2/2/4	3/4/0	349	31.00
dk16	27	3	390	95.50	4/8	2/1	454	39.60
planet	48	19	849	140.40	3/4/2/6	8/6/5/0	978	24.10
pma	24	8	385	62.30	2/2/2/4	1/3/4/0	400	34.30
indust1	94	8	606	74.00	4/4/12	3/5/0	783	25.80
indust2	67	1	137	35.20	10/11	1/0	193	17.50
indust3	33	15	1070	101.00	4/4/6	7/8/0	1325	36.10

compared with the previously reported data using two-way decomposition schemes of [3] and [14]. Our program gives lower delay than those in [14], and lower area in the majority of the tested examples. The method of [3] gives slightly better area results for two-way decomposition. However, delay estimates were not available for comparison. It should be noted that the primary goal in [3] was to reduce the area. In our approach, both the area and the delay are considered. As the number of submachines obtained with our method is usually larger than two it can be expected that the delays reported here are better than those using the method of Ashar *et al.* [3]. The data obtained could not be compared with other results on multiway FSM decomposition since none has been published to date.

5. VERIFICATION

We now formally define the verification problem. Given two descriptions of a sequential circuit and the correspondence between their reset states, determine if the two machines represent the same functionality. The first machine, or the specification, is described by a state transition table in symbolic form; the other machine can be in symbolic, encoded or encoded and optimized form. Thus, we really have three problems here: (1) verification of the decomposition process; (2) verification of the encoding process; and (3) verification of the synthesis/optimization process. Our general framework can readily handle all three problems.

5.1 The Verification Procedure

In a typical approach to the verification of FSM networks the state space of the network is represented as a product of the states of the constituent FSMs (submachines) [13, 24]. As a result the state space of the decomposed machine may become much larger than that of the prototype machine, as it may have several unused states that do not correspond to the states of the prototype machine. Checking the increased state space of the decomposed machine may significantly increase the complexity of the verification process. We use an efficient method to restrict the verification search to the state space of the prototype specification only, thus making the verification process more efficient. We use the information present in the prototype machine at the symbolic level to verify the decomposed machine network at the behavioral as well as at the logic level. We assume that the prototype machine has the correct specification and we verify the correctness of the decomposed machine with respect to that specification. A simple look-up table maintains the mapping of the states of the prototype machine to the states of the submachine. This helps in the verification process by allowing the system to check the output as well as the next state of the network under test, reducing the verification time.

There are two possible situations in the verification of an FSM network, depending whether the network was obtained from the decomposition of a single FSM, or generated directly from a high level description. When the FSM network is obtained as a result of FSM decomposition, both the output distribution (the partitioning of the primary outputs of the prototype machine among the submachines), and the state decomposition information, or state mapping, are known (see Section 4). We define the *state mapping* as the information about partitioning of the original states of the prototype machine in the process of creating the internal states of the individual submachines. It defines the mapping of the internal states of each submachine to the states of the prototype machine, and is captured in the *state map table* constructed as part of the verification process. When the submachines are state-minimized (have no equivalent states), each entry of the state map table has exactly two columns. For each row of the table the first column contains the state of the prototype machine; the second one contains the state of the decomposed machine represented as a product of the corresponding states of each of the submachines that correspond to that original state (see Example 3 below). Each product will have as many elements as the number of submachines, with one state name for each submachine. The procedure to build the state map table when state mapping is not known is given in the next section.

Example 3: Consider a prototype machine M with 4 states, $S = (A, B, C, D)$ as shown in Table III.

Suppose it is composed of two submachines, M_1 and M_2, defined by the following partitions: $\pi_1 = \{A, B : C, D\} = (\alpha : \beta)$ and $\pi_2 = \{A, C : B, D\} = (\gamma : \delta)$. This partition information is captured in the following state map table, Table IV.

According to this state mapping, state A of M maps to state α of M_1 and to state γ of M_2, so that when the prototype machine is in state A, the decomposed machine should be in state (α, γ), etc.

In the case of FSM networks generated directly from high level description, only output distribution is assumed to be known. Such networks may be obtained directly from a VHDL specification, where each submachine corresponds to a VHDL process. The decomposition is implicit in the specification but the state mapping is typically not available.

Our verification algorithm is general enough to work in both situations: it can verify the correctness of the FSM decomposition, or the FSM network obtained from a high-level specification. It requires the information about the output distribution, and, optionally, the state mapping, to be provided in a special *decomposition file*.

TABLE III
Prototype Machine for Example 3

	0	1
A	A, 01	D, 11
B	C, 00	A, 01
C	C, 00	B, 10
D	A, 01	C, 00

TABLE IV
State Map Table for Example 3

A	α, γ
B	α, δ
C	β, γ
D	β, δ

The output distribution information is necessary to reconstruct the output vector and to verify the output of the decomposed machine. The state mapping may or may not be provided. If it is provided, the state map table is built as shown in Example 3; otherwise the table will be automatically constructed during the verification process, as described in Section 5.3. The data in the decomposition file, together with the state transition information, provided in the *submachine file* for each submachine, allow the algorithm to verify the output as well as the next states of the decomposed machine.

The verification tool can be used at various stages of the synthesis process. To verify a system when the state mapping is known and both the prototype and the decomposed machines are in a symbolic form, we begin by building the state transition graph of the prototype machine. Each edge of the graph is then traversed in a depth first fashion and the corresponding input is applied to the decomposed machine for simulation. Since the state mapping is known, the state of each submachine, corresponding to a given state of the prototype machine, is readily determined. This allows to compare the next state of the decomposed machine with that of the prototype machine. The outputs of the decomposed machine are also compared with the outputs of the prototype machine. The decomposed machine is said to be equivalent to the prototype specification if every edge of the state transition graph is verified for the output as well as for the next state. The details of the graph traversal technique are explained in the following section.

5.2 Enumeration-Simulation Technique

The verification of the decomposed machine is based on a depth-first traversal of the STG of the prototype machine. First, the STG of the prototype machine is built, and the state map table is created. The verification algorithm begins with both the prototype and the decomposed machines being in their reset states. The procedure checks to determine if all fanout edges from the current state have been enumerated. If so, no further enumeration is necessary and the algorithm backtracks to the previous state. Otherwise, the next step is to enumerate the fanout edges from the current state. An applicable input vector for the present state of the prototype machine is formed based on the transition edge selected for traversal. The next state is obtained by checking the fanout node of the selected edge. This state then becomes a present state for the next iteration of the algorithm. The corresponding input vector is then applied as the simulation input to the decomposed machine, and the next state and outputs are obtained.

To verify the correctness of each transition, we check if the output of the decomposed machine implies the output of the prototype machine. If the two outputs are identical then we check if the next state of the decomposed machine matches the next state of the prototype machine according to the mapping in the state map table. This procedure is then repeated, and another edge of the STG of the prototype machine is enumerated. Search along a particular path is terminated when all the fanout edges from that state have been verified. At this stage the algorithm backtracks to the previous state. The decomposed machine is said to be equivalent to the prototype machine if every state and output are checked correctly for every transition of the STG. The verification of the outputs requires the correct merging of the output bits from the submachines using the output distribution information.

Example 4: Consider again machine M and its two-machine decomposition defined in Example 3. Fig 6 shows the STGs of the prototype machine and both submachines. Suppose that the reset state of the prototype machine is A, and the reset states of M_1 and M_2 are α, and γ, respectively (notice that $\alpha \cdot \gamma = (A, B) \cdot (A, C) = A$). Our enumeration-simulation algorithm will begin from the reset state of each machine; the prototype machine will be in state A while the decomposed machine will be in state (α, γ). Suppose we first traverse the edge corresponding to the loop on state A in the prototype machine by applying input vector 0. If we apply the same input to both submachines, the state loops back to state α in M_1, and to state γ in M_2, i.e., to state (α, γ) in the decomposed machine. We verify the output bits to be 01 for both machines, thus verifying this transition edge.

Then we pick the next untraversed fanout edge from state A; this corresponds to the edge $A \rightarrow D$ and input vector 1. Applying this input to both submachines takes M_1 to state β and M_2 to state δ. The generated output vector, 11, verifies correctly with the prototype output. We now verify the next state. Since $\beta = (C, D)$ in M_1 and

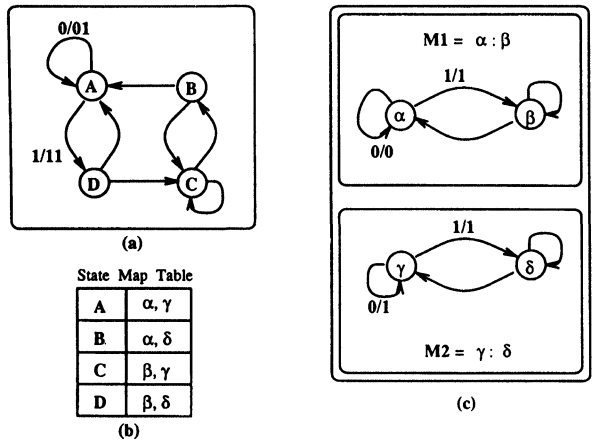


FIGURE 6 Example showing possible traversal path (a) prototype machine STG (b) state map table (c) decomposed machine STG.

δ = (B, D) in M_2, the next state of the decomposed machine is $\beta \cdot \delta$ = D, which is the same as for the prototype machine. The algorithm continues by examining the next edge.

Suppose we now pick the edge D $\rightarrow$ A to verify it. At this point we are back at state A and we have already checked all fanout edges of A; we backtrack to D and pick another fanout edge of D. We continue verifying the transition edges till we have covered all edges of the STG. A possible path to traverse all the edges is ($A \rightarrow A \rightarrow D \rightarrow A - D \rightarrow C \rightarrow C \rightarrow B \rightarrow C - B \rightarrow A$).

The main steps of the algorithm are shown in Table V. The procedure **traverse_DFS** performs the traversal of the STG as described above. The procedure verifies the two machines by traversing every edge of the STG only once. This is in contrast with some other enumeration-simulation approaches where every path (rather than edge) of the STG has to be traversed [11]. Thus the complexity of our algorithm is in $\Theta(E + N)$, the complexity of the depth-first search. Here, E is the number of transition edges and N the number of nodes in the STG. For comparison, the complexity of the approach in [11] is in $\Omega(E + N)$. Our approach is more efficient because we verify the output as well as the next state of each transition of the machine under test, while the other approach verifies only the outputs in each path of the machine. The verification of the next state at each transition removes the need for traversing every path in the STG. It becomes sufficient to traverse every edge of the STG only once to verify the entire machine.

5.3 Generation of the State Map Table

A key feature of our system is the generation of the state mapping for the decomposed machine even if it is not explicitly provided in the decomposition file. This allows the system to verify FSM networks obtained from high level specifications, where such a mapping is not known. In this case the system first finds a mapping of the prototype states to the submachine states and constructs the state map table. This is done by traversing the state transition graph of both the prototype machine and the decomposed machine. The reset states of both machines form the first entry of the table. The prototype and the decomposed machines are taken to their reset state. An edge of the state transition graph of the prototype machine is then selected and the corresponding input is applied to the decomposed machine. The next state of the decomposed machine, which is the product of the corresponding next states of all the submachines, gives the required mapping for the next state of the prototype machine and is added to the state map table. By traversing different edges of the STG, the table is completed for every state of the prototype machine.

Example 5: Consider the machine decomposition defined in Example 4, but assume that the state mapping is not provided. The state table is constructed based on the state transition information and the reset states of the individual submachines using the procedure described above. Using the same reset states as before, A = α, γ forms the first entry of the table. Transition edge $A \rightarrow D$ on M, corresponding to the transitions $\alpha \rightarrow \beta$ on M_1 and $\gamma \rightarrow \delta$ on M_2, creates the second entry for the table: D = β, δ. Other entries are created in the same fashion by traversing the remaining edges.

It is often possible to obtain a decomposition, or create an FSM network, in which some of the submachines are not state-minimized and as a result have equivalent states. Our verification procedure using state map table

TABLE V
Main Verification Procedure Using Enumeration

```
/* M_p and M_d are the machines to be verified */
/* M_p being the prototype and M_d the decomposed */
/* M_d is composed of submachines M_i */

Procedure Verify(M_p, M_d)

    1. build_STG (Prototype);
    2. if decomposition known
           read_decomposition( );
       else
           build_map_table( );
    3. Reset both machines
    4. traverse_DFS (Prototype);
    5. for all (submachines(M_i)) simulate (M_i);
    6. build_output_word( );
    7. flag_1 = compare_outputs( );
    8. flag_2 = compare_states( );
    9. if! (flag_1 && flag_2) machines are not equivalent, exit( );
   10. continue DFS

endProc
```

needs to be modified to check for possible state equivalence. The state equivalence for a machine can be readily detected by examining its state transition graph. Consider the case when, during the graph traversal, the current state of the decomposed machine (say ϕ, γ) does not match the state in the state map table (say α, γ). In this case one must check for the equivalence between the corresponding states (here α and ϕ of the first submachine). If they are equivalent then additional mapping (in this case ϕ, γ) is added to the corresponding row of the state map table, see Fig. 7 (a), (b). This way the state map table may contain more than one state of the decomposed machine for every state of the prototype machine. If the corresponding states of the decomposed machine are not equivalent, the decomposed machines and the prototype are declared to be different. The procedure for creating a state map table is illustrated by the following example.

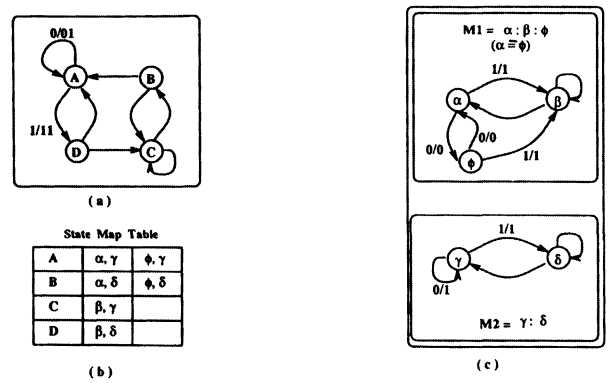


FIGURE 7 (a) prototype machine STG (b) state map table (c) decomposed machine STG with equivalnet states in submachine M_1.

Example 6: Consider the prototype machine M from the previous example but with a different decomposition. Submachine M_1 = (α: β: ϕ) has two of its states α, ϕ equivalent; see Figure 7(c). As before, suppose that the reset state of the prototype machine is A, and for the decomposed machine it is (α, γ). This creates the first entry in the table: $A = \alpha,\gamma$. Starting with the machines in their reset states and applying input 0 causes the prototype machine to stay in state A while the decomposed machine to go to state (ϕ, γ). This causes a conflict in our state map table in the sense that the first symbol (ϕ, corresponding to machine M_1) does not agree with the state name (α) in the table. According to the existing entry in the table at this stage of the algorithm, submachine M_1 should have been in state α; instead, it went to state ϕ. However, before declaring an error, the algorithm will check for the equivalence between states α and ϕ in submachine M_1. Since the two states happen to be equivalent, a new entry (and a new column) is created for the row corresponding to state A in the state map table, as shown in Figure 7(b). The algorithm will continue in a similar fashion to complete the state map table for the remaining states of the prototype machine.

Once the state map table is completed for all the states of the prototype machine the previously described verification procedure is followed. The only difference is that now each state of the prototype machine may correspond to more than one state of the decomposed machine. The prototype state should match at least one of those equivalent states, and the two machines should produce identical outputs. Otherwise, the prototype and the decomposed machines are declared to be different.

5.4 Verification of Encoded Machine

The next problem in our framework is to handle the verification at a *binary level* after the submachines have been encoded. This will in effect verify the encoding process. The problem of verifying an encoded machine is actually no different from verifying a machine at the symbolic level described in the previous sections. An encoded (but non-optimized) fully specified machine can be considered as a machine at the symbolic level since the codes of the states can be viewed as symbols representing the states. Since there are no don't cares, every state has a unique symbol represented by its binary code, and the previously described verification method can be readily used to verify the encoded decomposed machine.

The verification of the *optimized decomposed machine* (or the network of *optimized FSMs*) implies the verification of the synthesis/optimization process. We solve this verification problem by comparing the binary cover of each optimized submachine with the cover generated for that submachine from the prototype specification.

The following procedure is used to generate the *prototype on-set cover*, $C^{ON}_{prot}(M_i)$, for submachine M_i. For each transition edge in the prototype machine a binary cube representing this transition is created and added to the cover of the corresponding submachine. The output distribution is used to determine the submachines to which the particular cube belongs. The input field of the cube is the concatenation of the primary input bits and the binary code of the present state of that submachine. The output field of the cube is the concatenation of the next state code and the output bits assigned to that submachine. The encoding of the states for each submachine is obtained from the submachine files provided as input to the program. Similarly the *prototype off-set cover*, $C^{OFF}_{prot}(M_i)$, is created for M_i from the prototype specification. The construction of the two covers is illustrated by the following example.

Example 7: Consider the prototype machine M of example 3. Suppose we assign the following codes to the states of the submachines.

$$\begin{array}{ll} M_1 & M_2 \\ \alpha = (A, B) = 0 & \gamma = (A, C) = 0 \\ \beta = (C, D) = 1 & \delta = (B, D) = 1 \end{array}$$

The encoded tables of the two submachines are shown in Table VI.

The following on-set and off-set covers are obtained for the two submachines.

$$C^{ON}_{prot}(M_1) = \{1AD1, 1CB1\}$$
$$= \{10011, 10101\}$$
$$C^{OFF}_{prot}(M_1) = \{0AA0, 0BC0, 0CC0, 0DA0, 1BA0, 1DC0\}$$
$$= \{00000, 01010, 00110, 01100, 11000\}$$
$$C^{ON}_{prot}(M_2) = \{0AA1, 0DA1, 1AD1, 1BA1\}$$
$$= \{00001, 01101, 10011, 10101\}$$
$$C^{OFF}_{prot}(M_2) = \{0BC0, 0CC0, 1CB0, 1DC0\}$$
$$= \{00100, 01000, 11010, 11100\}$$

Let the on-set and the off-set covers of the *optimized* submachine M_i be denoted as $C^{ON}_{OPT}(M_i)$ and $C^{OFF}_{opt}(M_i)$, respectively. The decomposed machine is said to be equivalent to the prototype machine, if, for each submachine M_i in the decomposed machine network, the following conditions are satisfied

$$C^{ON}_{prot}(M) \subseteq C^{ON}_{opt}(M_i) \text{ and } C^{OFF}_{prot}(M) \subseteq C^{OFF}_{opt}(M_i) \quad (4)$$

The condition $C^{ON}_{prot}(M) \subseteq C^{ON}_{opt}(M)$ implies that every time the prototype machine has a 1 at its output, the submachine also has a 1. Similarly, the condition $C^{OFF}_{prot}(M) \subseteq C^{OFF}_{opt}(M)$ implies that the every time the prototype machine has a 0 at its ouput, the submachine output is also a 0. The satisfaction of both of these conditions implies that the submachine is equivalent to its corresponding unoptimized submachine obtained from the prototype specification. Clearly, the on-set of the optimized submachine must cover the on-set obtained from the prototype without intersecting the off-set; the two covers may not have identical on-sets if they have different don't-care sets, hence the covering relation. Our program uses Espresso routines to compare the binary covers. Recall that we are checking for the functional correctness of the decomposed machine with respect to the prototype machine and not the equivalence of the on-sets of the two machines, hence the sufficiency of the above condition. If all submachines are found to be equivalent to their unoptimized specifications, the optimized decomposed machine and the prototype machine are declared equivalent.

At this stage we are only concerned with the verification of the submachine optimization and assume that the decomposition is legal. The verification of the decomposition itself is readily accomplished by verifying the prototype covers of the submachines.

TABLE VI
Encoded Table for the Submachines in Example 7

M_1					M_2				
I	S_2	S_1	S_1'	Q_1	I	S_1	S_2	S_2'	Q_2
0	0	0	0	0	0	0	0	0	1
0	1	0	1	0	0	0	1	0	0
0	0	1	1	0	0	1	0	0	0
0	1	1	0	0	0	1	1	0	1
1	0	0	1	1	1	0	0	1	1
1	1	0	0	0	1	0	1	0	1
1	0	1	0	1	1	1	0	1	0
1	1	1	1	0	1	1	1	0	0

5.5 Verification Tool

The verification technique described in this section have been implemented as a program *VerSim*. Together with the decomposition tool and other synthesis tools it is a part of an interactive FSM synthesis system. The inputs to the program are: the *prototype file*, which contains the state transition table of the prototype machine in symbolic format; the *decomposition file*, with the output distribution information and, optionally, state mapping; and a set of *submachine files*, with the state transition table in symbolic Kiss-like format for each submachine. The verification version of the program is invoked with one of the following options: *-d* to perform the verification with known decomposition, *-u* with unknown decomposition, and *-c* to verify an optimized machine network.

Apart from verifying the decomposed machine with respect to the prototype machine, the program has several other options which may help the designer in the synthesis of an FSM network. These options and other useful features are briefly described below.

5.5.1 Errors checked—The verification tool checks for three basic types of errors and displays the appropriate error message. The first type of errors are simple file *system errors*. The system also checks for syntax errors in the description of the files. The program may terminate in the event of a syntax error. The second type of errors checked are the *decomposition errors*. The verification tool checks for possible errors in the decomposition specified in the decomposition file. Some of these errors may also be detected while verifying the submachine files. The detected errors include a decomposition

which will put the network in an unknown state or in more than one state. This can happen if the product of the decomposition is not a zero-product, $\pi(0)$, implying an illegal decomposition. The third type of errors checked by the program are *transition errors*. The tool starts by verifying the individual submachine files. Transitions to wrong next states and assertion of wrong outputs are reported. It also reports any missing transitions, i.e., transitions which are present in the prototype machine but missing in the submachines. Checking for this error is particularly important because the prototype and the decomposed machines are said to be equivalent only if every transition present in the prototype is also present in the decomposed machine.

5.5.2 Simulation—The simulation option of the system is invoked by the *–s* option of the program. It allows the user to observe the outputs and the next states of the decomposed machine for a given input, by exercising simultaneously the state transitions of the submachines. The simulation is controlled using various commands, which allow the user to set the input, bring the states to their declared reset states, print the current state and the output vector, log the current simulation status, etc.

5.5.3 Submachine generation—The generation tool option, invoked by the *–g* option on the command line, creates the submachine files from a given prototype file as well as the state mapping. The format of the prototype file and the decomposition file are the same as for the verification option described above. This option can be used to experiment with different decompositions. Given a decomposition, it checks if it is legal and generates the set of the submachine files.

5.6 Verification Results

The verification program has been tested on several MCNC examples and some industrial circuits. All circuits were decomposed using the decomposition technique described in Section 4. The program successfully verified all the examples shown in Table VII using all the verification options. The first four columns in the table give the circuit statistics: the number of inputs, states, transition edges, and outputs. The column labeled "# sub" specifies the number of submachines in the decomposed machine. The last two columns report the CPU time (in seconds) needed to verify the system using the '*–d*' and '*–u*' option on a DECstation 5000/125 machine. In all tested examples the *-u* option (when the state mapping is unknown) required longer time to verify the machines because of additional step needed to build the state map table.

6. CONCLUSIONS

We have presented in this paper a new, efficient multiway decomposition technique for finite state machine decomposition, and a powerful verification/simulation scheme for verifying the resulting decomposed machine networks. Both techniques, implemented as complete C programs, are part of an interactive FSM synthesis system developed at the University of Massachusetts at Amherst. This system also includes tools for synthesis of individual machines, encoding of the FSM network, and floor-planning and schematic viewing. It is designed to serve as a useful interactive development tool for control dominated applications.

TABLE VII
Verification Results

Name	inputs	states	trans edges	outputs	# sub	$time^d$	$time^u$
indust1	6	94	233	8	3	0.98	0.98
indust2	8	67	585	1	2	2.05	2.35
indust3	12	33	793	15	3	5.21	8.18
indust4	11	16	1804	15	4	33.0	65.4
bbara	4	10	60	2	3	0.08	0.10
cse	7	16	91	7	2	0.11	0.16
pma	8	24	73	8	4	0.18	0.25
dvram	8	35	47	15	4	0.15	0.18
planet	7	48	115	19	4	0.45	0.48
dk16	2	27	108	3	2	0.15	0.16
ram_test	16	7	117	24	4	0.48	0.52
scf	27	121	166	56	3	0.89	1.03
sse	7	16	56	7	3	0.10	0.11
styr	9	30	166	10	4	0.53	0.78
sand	11	32	184	9	4	0.63	0.93

Our decomposition program emphasizes the delay reduction. We have demonstrated substantial performance improvement in the resulting decomposed machines obtained by our technique. We have shown the potential and efficiency of our system to verify the decomposed machine networks with respect to their prototype specifications. The programs were used to decompose and verify in reasonable time machines with up to 1800 edges and up to 121 states. The verification program is capable of verifying even larger machines; however, we could not demonstrate this for lack of larger examples available to us. Since there are no standard benchmarks for verification, it is hard to compare the efficiency of our algorithm with other methods. The verification tool can be used at various stages of the synthesis process to check the correctness of each synthesis step. An important application of this program is the verification of FSM networks derived directly from high level (VHDL) specifications, when the decomposition information is not explicitly available.

Acknowledgements

The authors wish to thank Maya Yajnik for providing in-depth analysis and implementing the decomposition algorithm presented in the paper.

References

[1] D.B. Armstrong, "A programmed algorithm for assigning internal codes to sequential machines," *IRE Trans. Electron. Computers*, vol. EC-11, no. 4, pp 466–472, August 1962.

[2] P. Ashar, S. Devadas, and A.R. Newton, "A Unified Approach to the Decomposition and Re-decomposition of Sequential Machines," *27th ACM/IEEE Design Automation Conference*, pages 601–606, June 1990.

[3] P. Ashar, S. Devadas, and A.R. Newton, "Optimum and heuristic algorithms for an approach to finite state machine decomposition," *IEEE Trans. Computer-Aided Design*, vol. 10, pages 296–310, March 1991.

[4] R. Brayton, R. Rudell, A.S. Vincentelli, and A. Wang, "MIS: A multiple-level logic optimization system," *IEEE Trans. Computer-Aided Design*, Nov 1987, pp 1062-1081.

[5] M.C. Browne, E.M. Clarke, D.L. Dill and B. Mishra, "Automatic verification of sequential circuits using temporal logic," *IEEE Trans. Computers*, vol. C-35, pages 1035–1044, December 1986.

[6] R.E. Bryant, "Symbolic verification of MOS circuits," *Proc. of the 1985 Chapel Hill Conference on VLSI*, pages 419–438, December 1985.

[7] J.R. Burch, E.M. Clarke, K.L. McMillan and D.L. Dill, "Sequential circuit verification using symbolic model checking," *27th ACM/IEEE Design Automation Conference*, pages 46–51, June 1990.

[8] O. Coudert, C. Berthet and J.C. Madre, "Verification of sequential machines using symbolic execution," *Proc. of the Workshop on Automatic Verification Methods for Finite State Machines*, 1989, Grenoble, France.

[9] S. Devadas, "Optimizing Interacting Finite State Machines Using Sequential Don't Cares," *IEEE Trans. Computer-Aided Design*, pages 1473–1484, December 1991.

[10] S. Devadas, A.R. Newton, "Decomposition and factorization of sequential finite state machines," *IEEE Trans. Computer-Aided Design*, vol. 8, pp 1206-1217, Nov 1989.

[11] S. Devadas, H-K. T. Ma, and A.R. Newton, "On the verification of sequential machines at differing levels of abstraction," *IEEE Trans. Computer-Aided Design*, pages 713–722, June 1988.

[12] A. Ghosh, S. Devadas, and A.R. Newton, *Sequential Logic Testing and Verification*, Kluwer Academic Publishers, 1992.

[13] G.D. Hachtel and R.M. Jacoby, "Verification Algorithms for VLSI Synthesis," *IEEE Trans. Computer-Aided Design*, pages 616–640, May 1988.

[14] Z. Hasan and M.J. Ciesielski, "FSM decomposition for performance optimization," *Technical Report TR-91-CSE-14*, Department of Electrical & Computer Engineering, University of Massachusetts, Amherst, 1991.

[15] Z. Hasan, D. Harrison and M.J. Ciesielski, "A fast partitioning method for PLA-based FPGAs", *IEEE Design & Test of Computers*, pp 34–39, December 1992.

[16] S.H. Hwang and A.R. Newton, "An efficient verifier for finite state machines," *IEEE Trans. Computer-Aided Design*, pages 326–334, March 1991.

[17] J. Hartmanis and R.E. Stearns, *Algebraic Structure Theory of Sequential Machines*. Englewood Cliffs, NJ. Prentice-Hall, 1966.

[18] L. Jóźwiak, "Simultaneous decomposition of sequential machines," *Microprocessing and Microprogramming*, North-Holland, Vol 30, 1990, pp 305–312.

[19] L. Jóźwiak and J.C. Kolsteren, "An efficient method for the sequential general decomposition of sequential machines," *Microprocessing and Microprogramming*, North-Holland, Vol 32, 1991, pp 657–664.

[20] L. Jóźwiak, "Decompositional logic synthesis: Correctness Aspects", *Proc. of the First Asian Pacific Conference on Hardware Description Languages, Standards & Applications*, Brisbane, Australia, Dec 1993, pp 55–64.

[21] B.W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs", *Bell System Technical Journal*, Vol. 49, Feb. 1970, pp 291–307.

[22] Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill, 1978.

[23] G. Saucier, "State minimization of asynchronous sequential machines using graph techniques," *IEEE Trans. Computing*, vol. C-21, pp 282-288, March 1972.

[24] K.J. Supowit, and S.J. Friedman, "A new method for verifying sequential circuits," *Proc. 23rd Design Automation Conference*, pages 200–207, June 1986.

[25] J.J. Shen, Z. Hasan and M.J. Ciesielski, "State Assignment for General Finite State Machine Networks", *Proc. of EDAC '92*, Brussels, Belgium, March 1992, pp 245–249.

[26] M. Yajnik and M.J. Ciesielski, "Finite state machine decomposition using multiway partitioning," *Proc. Int'l Conference on Computer Design*, 1992.

[27] C. Yeh and C. Cheng, "A General Purpose Multiple Way Partitioning Algorithm", *28th ACM/IEEE Design Automation Conference*, 1991.

Biographies

ZAFAR HASAN is a Doctoral student in the Electrical and Computer Engineering Department at the University of Massachusetts, Amherst, and a research assistant in the VLSI Design Laboratory. His research interests include CAD for VLSI design, logic synthesis and computer architecture.

Zafar received a B.Tech degree in electrical engineering from Banaras Hindu University, Varanasi, in 1989, and an M.S. in electrical and computer engineering from the University of Massachusetts, Amherst in 1992. He is a student member of the IEEE. He can be contacted through internet at *hasan@spock.ecs.umass.edu.*

MACIEJ CIESIELSKI is Associate Professor of Electrical and Computer Engineering at the University of Massachusetts, Amherst. His research focuses on high-level and logic synthesis, and performance optimization of VLSI systems. In 1993/94 he was on sabbatical at ENST Paris, working on architectural optimization for DSP, and at INPG Grenoble, working on controller synthesis. He has served on the technical program committees for ICCAD, ICCD, and IWLS. He is a member of the IEEE Computer Society.