Decomposition & Functional Verification of FSMs[1]

Zafar Hasan *Maciej J. Ciesielski*

Department of Electrical & Computer Engineering
University of Massachusetts
Amherst, MA 01003

Abstract

In this paper 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 and simulation of the FSM network at different levels of abstraction. The problem of decomposition is solved using a multiway graph partitioning technique. The number of submachines is determined dynamically during the partitioning process. The verification algorithm is used to verify the correctness of the FSM network at any stage of the synthesis process. It can be used to verify (a) the result of FSM decomposition on a behavioral level, (b) the encoded FSM network, and (c) the 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 in a depth first fashion and simulation of the decomposed machine network. Both the decompostion and verification/simulation algorithms have been implemented as a computer system and tested on a set of benchmark examples.

[1]This work was supported by a grant from NSF under contract number MIP 9013013.

1 Introduction

Sequential circuits play a major role in the control part of digital systems and efficient tools for their design are essential for computer-aided design of VLSI chips. 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 Field Programmable Gate Array (FPGA) or Programmable Logic Device (PLD) technologies are used for final implementation. The logic in FPGA and PLD technologies 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 [14]. 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 the 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 a network 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. The decomposition approach presented here aims at optimizing the performance of the resulting implementation. It is unique in the sense that 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 both the outputs and the internal states of the submachines.

Several algorithms have been proposed to decompose an FSM into two interacting submachines [16], [9], [3], [13], but no significant results have been achieved in the field of multi-machine decomposition. Here we present an algorithm to decompose an FSM into an arbitrary number of submachines. The primary goal of this method is the improvement of performance and/or the area of the final 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. 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. [16], [9]). 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. We use a better 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 error 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. 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 unoptimized one [11]. 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. There are very 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, thus 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 [20], [10]. Our verification/simulation algorithm has been implemented in a system called $VerSim$. It can be used for formal design verification after decomposition of the prototype machine, or for simulating the design. Implementation verification can be done after encoding the submachines. It can also be used for logic verification of the optimized submachines. The tool has an additional submachine generation option to help FSM network designers. This option can generate the submachines from the prototype machine given a valid decomposition of the states of the prototype machine. 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 also be used to verify FSM networks obtained from high level specifications.

The rest of the paper is organized as follows. Section 2 describes the definitions and notation. Section 3 breifly reviews some of the earlier decomposition and verification methods. Section 4 contains the details of the decomposition method. Section 5 describes our verification algorithm in detail with section 5.3 describing the enumeration-simulation technique. The other subsections describe the application of the algorithm to the various different cases which we have covered in this paper. Section 6 describes the simulation tool and some of the additional helpful features of the $VerSim$ system. Section 7 contains an overview of the $VerSim$ system describing the format of the input files. The last section presents some

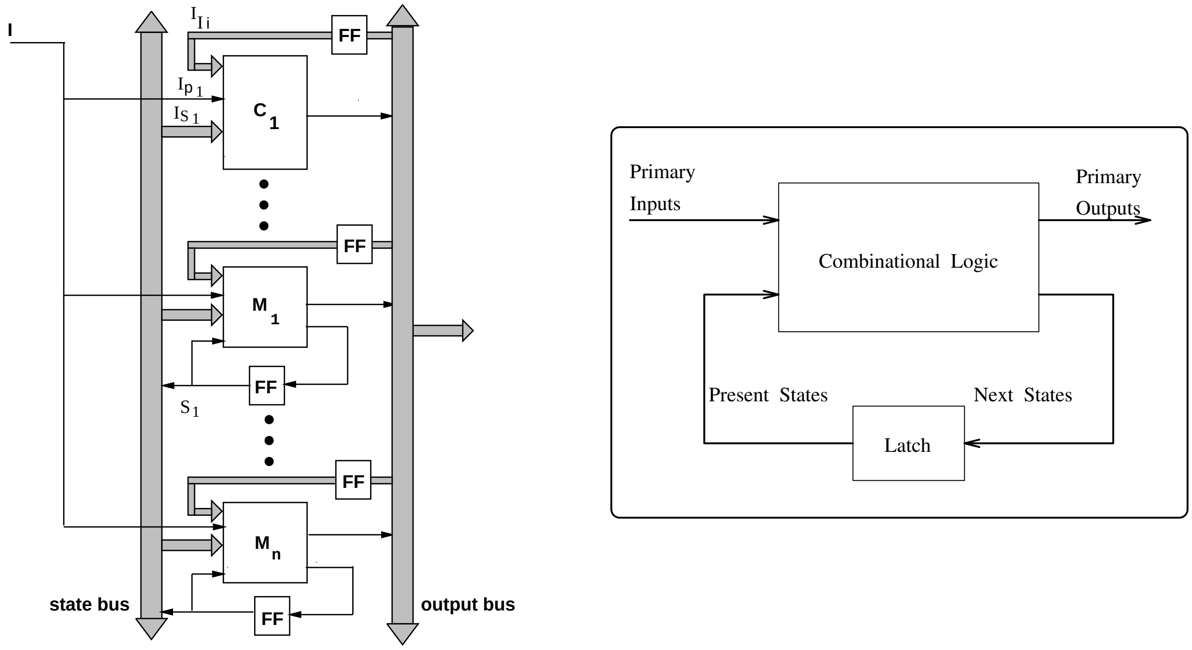


Figure 1: General decomposition topology and general sequential circuit

results and conclusions on the verification method.

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)**.

Definition 2.1 *A* **partition** π *on a set of states* S *is a collection of disjoint subsets of* S*, called 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 *[18]. 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 such a network is shown in Fig. 1.

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 $\pi_P = \pi_1.\pi_2 = \{A : B : C : D\}$. Here the first block of π_P was obtained by taking the intersection of $\{A, B\}$ and $\{A, C\}$. Similarly the other blocks are also obtained.

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 $\pi_1, \pi_2, ..., \pi_n$ such that

$$\pi_1.\pi_2...\pi_n = \pi(0)$$

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

A legal decomposition satisfies the behavior of the original 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 problem of verifying the equivalence of two implementations of a sequential circuit and the problem of test generation for such circuits are intimately related. 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. A general sequential circuit is shown in Fig. 1(b). 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 equivalence, 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 [18]. 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* **k-equivalent**. *If a finite differentiating sequence does not exist for a state pair, the states in the pair are said to be* **equivalent**.

Two states are said to be **output-equivalent** or **single-cycle equivalent** if they are 1-equivalent [18].

3 Previous Work

The decomposition of sequential machines was first treated in a formal way by Hartmanis and Stearns in [16]. They proposed two types of decomposition, parallel and cascade, based on the topology of the decomposed machine.

One of the simplest ways in which a machine can be broken up into submachines is the **parallel decomposition**. If the decomposed machine has 2 submachines $M1$ and $M2$, then their structure is such that they 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 again the overall realization has been broken up into two submachines, each driven by the same input sequence. But the two machines do not operate independently. Submachine M_2 is supplied, by means of auxiliary inputs, with information about the current internal state of M_1. This information influences the state transitions that M_2 makes and enables M_2 to generate the appropriate output sequence. The possibility of passing state information from M_1 to M_2 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 only.

Another form of decomposition was presented by Devadas and Newton in [9]. Here 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 a definite cost function to optimize and does not guarantee anything about the quality of the decomposition.

Ashar *et al.* in [3] presented another decomposition method where the topology of the decomposed machine is a general decomposition topology as shown in Fig. 1. Optimum and heuristic algorithms for the general decomposition of FSMs are shown, such that the total number of product terms in the one-hot coded and logic minimized submachines is minimum or minimal is shown. They use the number of product terms in the decomposed machine as their 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 that represents an optimum FSM decomposition algorithm, under the specified cost function.

All the above mentioned decomposition methods basically aim at two-way partition. There is no significant work done on multi-way partitioning of sequential machines. Although in principal, the general decomposition method described by Ashar *et al.* in [3] can be extended to multi-way partitioning, the method is not clearly indicated. Further, none of the published work aim at minimizing the delay of the decomposed machine.

The problem of verification of single FSMs has been under investigation for a long time. Some of these algorithms are based on the product of the two machines to be verified [12]. Some others are based on graph traversal, enumeration-simulation [10] and symbolic STG traversal [7]. Most previous 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 enumeration-simulation approach was first presented in [10]. In this approach all state transitions of one machine are enumerated and simultaneously simulated on the other machine. During 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 more efficient. A similar approach was presented in [15]. However, the system described in [15] does the verification of a single machine after it is encoded. We extend the idea to handle the verification of decomposed machines.

Recently, 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 [6] [7]. 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) [5]. These methods work best for circuits that have certain regularity in the structure of the STG. However, for certain circuits, the BDDs are so large that they cannot be built and stored in the computer. Also, for certain machines, the transition relations become very complex causing each iteration to take a long time. Further, the BDDs can be constructed only after the machines have been encoded and the method cannot be used for verification at the symbolic level. Thus 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, [4] [20]. 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. Our framework can verify a decomposed FSM at various levels of abstraction with its original prototype specification given at the symbolic level.

4 Decomposition

The problem of FSM decomposition can be formulated as that of assigning output bits to the submachines and obtaining a legal partition on the set of states for each submachine. The original states are grouped together in such a way that each block represents one internal state and is assigned a distinct code in this machine. However, each pair of states has to be differentiated in at least one submachine.

4.1 The Architecture

The architecture of the decomposed finite state machine is as shown in Fig. 1. We refer to this topology as the **general decomposition** topology since it can represent any 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 considered either symbolic or binary, depending on the initial specification.
2. A set of **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 **internal states**, S_i.

In principle, the encoding of each submachine can be arbitrary, as long as its internal states are given distinct binary codes. The primary inputs are made available to every submachine. The states of one machine are made available to the others if necessary. The outputs are generated directly by individual submachines. Note that state information is being shared by the submachines and that, in general, the submachines are not independent of each other.

Example 1: Suppose we have a prototype machine M with 3 primary outputs - Q_0, Q_1, Q_2 and 5 states $S = (A : B : C : D : E)$. Suppose this machine is decomposed into 2 submachines, M_1 and M_2 as follows:

$$\begin{array}{llll} M_1 : & States & (AC : BD : E) = (x_1, x_2, x_3) & Output = Q_1 \\ M_2 : & States & (ABE : CD) = (y_1, y_2) & Output = Q_0, Q_2 \end{array}$$

Machine M_1 has output Q_1, and states: x_1, x_2 and x_3. In this machine, original states A and C will get the same code. Similarly states B and D will get the same code. Machine M_2 has two internal states y_1 and y_2. In machine M_2, states, A, B and E will get the same code. Each submachine will now require the state information from the other submachine in addition to the primary input. Consider an entry in the STT of the original machine M.

Input	Present State	Next State	Output(Q1)
11	B	C	0

State B is assigned code x_2 in M_1 and code y_1 in M_2. The same entry in M_1 will now be represented by:

Input	Present State	Next State	Output(Q1)	
11	y_1	x_2	x_1	0

This shows that it is necessary to use the state information from other submachines to distinguish two states belonging to the same partition in a submachine. ∎

4.2 Decomposition Algorithm

The primary objective of our approach is to reduce the complexity of each submachine, while attempting to keep the total number of submachines small. The decomposition algorithm works in two main steps: (1) partitioning of outputs and (2) partitioning of the states. The partitioning algorithm presented here is different from most of the existing methods in this regard. Most of the earlier FSM partitioning algorithms 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 partitioning and assignment of the outputs among the submachine so as to achieve the best possible results.

The first step is to partition the output bits among the submachines. The adopted cost measure is the minimization of estimated performance and area. This is done based on a 2-level implementation strategy. We use the number of product terms as the measure of circuit complexity as this is the most accurate estimate possible based on symbolic minimization. The output partition determines the number of submachines in the final decomposition also. The next step is to partition the original states of the machine. This is done for each of the submachines individually. The state partition determines the internal states of the submachines based on the information of output partitioning derived in the previous step.

The problem of partitioning of the states is solved by constructing a special graph $G_S(V, E_S)$ where V is a set of nodes and E_S a set of weighted edges, and performing graph partitioning on it. V is in one-to-one correspondence with the states of the prototype machine. The weighted edges E_S in the graph represent the desired adjacency requirement among the states. The weight of each edge $e \in E_S$ can be either positive or negetive. A positive weight reflects the extent to which the number of product terms would be reduced, if the two states were given the same code in a particular 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" (negetive) 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.

Output bit partitioning is chosen as the first step because the assignment of outputs to a submachine significantly affects the way in which the number of product terms are reduced with a partition on the states. Thus state partitioning is done after the outputs of the submachine are known. The idea of repulsive edges is specially important in the second step where the partitioning of the states is sought. These repulsive edges have negative weights which indicate that they are the preferred edges of the cut in the multiway graph partitioning algorithm.

4.2.1 Partitioning of the Outputs

The cost of a particular 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 complexity cannot be

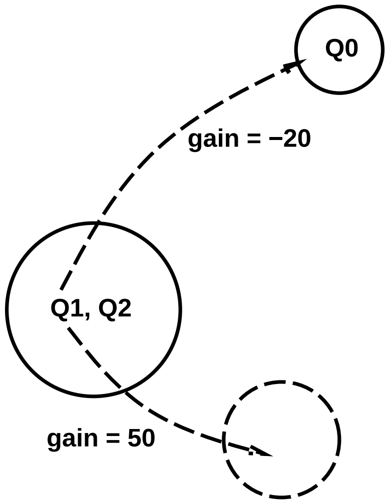


Figure 2: Possible moves for output bit Q_1

exactly determined without knowing the partitioning of the states or the final implementation. However it can be estimated by finding the number of product terms assuming 2-level implementation and one-hot coding of the states. Hence the total cost of the decomposition is the sum of the costs of all the submachines. The area cost of a submachine is estimated using the following equation

$$Area = (2 * I + O) * P)$$

where I is the number of inputs, O the number of outputs and P the number of product terms in the submachine.

The outputs bits of original machine are grouped together on the basis of maximum saving in area and performance. The following way of partitioning the outputs is proposed. Starting from a random partition, the cost is determined by finding the estimated number of product terms for each group in the partition. Each group of outputs will be assigned to one submachine in the final implementation. Taking one output at a time, the algorithm finds the best new group to which to assign that output. A group without any outputs can also exist. 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. This is explained by example 2 shown below. Occasionally, random moves are made to avoid the trap of a local minimum. There are several passes. In each pass, each output is moved only once. Once an output has been moved, it is locked for that entire pass. When an attempt has been made to relocate all the outputs, a new pass begins where each output is moved again.

Example 2: Suppose we have a prototype machine M with 3 primary outputs - Q_0, Q_1, Q_2 and 5 states $S = (A : B : C : D : E)$ (same as in example 1). Suppose the output partitioning step begins with a random partition with 2 submachines such that output Q_0 is in one group and Q_1, Q_2 are in the other group with a total cost of 100. Now we have two possible moves for Q_1. We can shift to the group with Q_0 or we can put it in a third group of its own as shown in Figure 2. The cost after the first move would be 120 while that after the second move is 50. The gains from the current position to the new partition are shown in the edges in the figure. Based on a higher gain with the second move the algorithm assigns Q_1 to a third partition. This decomposition will now have 3 submacines. ■

The number of submachines in the decomposition is determined from the partition of outputs obtained by the above procedure. An extra submachine, with no outputs may be necessary if all pairs of states do not have different codes in at least one of the submachines. With the knowledge of the output bits for each submachine the states can be then partitioned in the next step so as to minimize the number of product terms in that submachine.

4.2.2 Partitioning of the States

For each submachine, the states of the prototype machine are partitioned into *blocks* of states, each block forming an internal state of the submachine. To accomplish this, a weighted graph G_S is constructed with the nodes representing the states of the prototype machine. One such graph for each submachine is constructed and partitioned using the multiway graph partitioning algorithm described in this section. While constructing a graph for a submachine, the algorithm uses information about previously determined state partitions in other submachines.

An important issue is how the partition of states for a given submachine is influenced 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. The problem is handled by building on descriptions of submachines whose partitions have already been obtained. The edge weights of the graph G_S reflect the extent to which the product terms would be reduced if those two states were given the same code in the submachine being considered.

The construction of graph G_S can be briefly described as follows. We add a positive weight on the edges based on the following two rules. An edge with weight $w(e) = 1$ is added between the nodes corresponding to the states with the same next state. For example, if for certain input, states s_1 and s_2 produce state s_3 as their next state, then an edge is added between the nodes v_1 and v_2. If the edge already exists, the weight of the edge is incremented by one. An edge with weight $w(e) = 1$ is added between the next state nodes that are derived from the same present state. For example, if s_1 produces, under two different inputs, s_2 and s_3 as the next states, then an edge with $w(e_{23}) = 1$ is created.

A negative weight is added to the edge between nodes representing present states for which, for the same input, the output is different. This is needed to distinguishe these states in order to generate the output. Although any pair of states can be distinguished with the knowledge of other submachines' internal states, to reduce the communication costs it is advantageous that these states have different codes in this submachine. 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.

In	PS	NS	Out
11	A	C	10
11	B	C	10
00	D	E	11
00	A	B	00

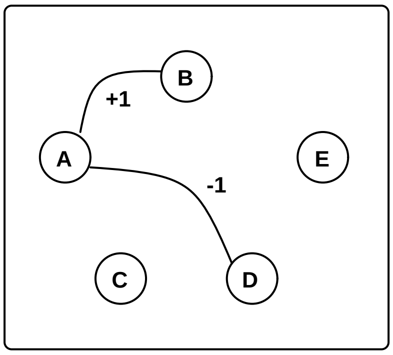


Figure 3: Example showing assignment of edge weights

The weights assigned to the edges of the state partition graph G_S 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], [19]. 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.

Example 3: Consider the partial truth table shown in Figure 3. State A and B go to the same next state for the same input. Hence we get an edge with a positive wieght between A and B in the adjacent graph shown. State A and D go to different states for the same input given so using the rule for negetive weights we get an edge with -1 between the A and D. ■

Multiway partitioning of the graph constructed in this way gives a grouping of states whose cost is minimum. The cost of the partition is defined as the sum of the weights on cut edges. 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.

The algorithm for multiway graph partitioning uses ideas from the Kernighan and Lin min-cut algorithm [17] for two-way graph partitioning. The graph partitioning algorithm is used to determine the internal states of a submachine. The goal is to create a set of partitions, whose product is equal to π_0. Each partition identifies one submachine, and its states are defined by the individual blocks of the partition. Consider the graph in Fig. 4. An optimum partition of this graph is $(CF : A : BDE)$. Starting from a random partition, the algorithm iteratively improves on it, reducing the cost function. Occasionally, the program makes a random move to make sure that it does not get trapped in a local minimum.

The maximum number of groups can be specified by the user. For a totally free partitioning the maximum number of groups is the number of nodes. Since empty groups are allowed, the actual number of non-empty groups may vary depending on the cost of the partitions. There are several passes of the algorithm. In each pass the following steps are taken.

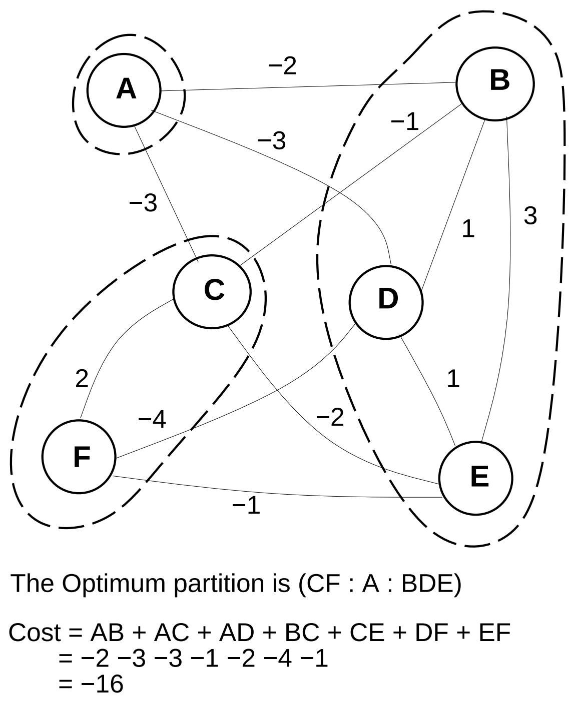


Figure 4: Multiway Graph Partitioning

- 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 negetive) 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 gain. 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.

4.3 Results

The method described in this section has been implemented as a C program and tested on examples from the MCNC benchmark suite. The current version of the program puts greater emphasis on improving performance than on area. The results for 2-level implementation are shown in Table 1. In the table, S represents the number of states, O the number of outputs and P the number of product terms. Techniques described in [21] were used to carry out the encoding of states in the communicating machines. Delay and area were calculated as described in [13]. The results for multilevel implementation are shown in Table 2. The multilevel implementation was done using MISII and the results shown are as reported by the program.

The results from both table indicate that the delay of the decomposed machine is always reduced. In most cases the area is also reduced. The data obtained for delay and area can be

Table 1: Experimental Results of 2-level implementation

Bench-mark	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
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 2: Expremintal results of multi-level implementation

Example	Prototype Machine		Decomposed Machine	
	Area	Delay	Area	Delay
bbara	97	30.30	106	14.00
cse	249	69.10	349	31.00
dk16	390	95.50	454	39.60
planet	849	140.40	978	24.10
pma	385	62.30	400	34.30
indust1	606	74.00	783	25.80
indust2	137	35.20	193	17.50
indust3	1070	101.00	1325	36.10

compared with previously reported data on the two-way decomposition schemes of [3] and [13]. Comparing the results reported here with those in [13] our program gives lower delay in all, and lower area in the majority of the examples tried. The method of [3] gives slightly better areas 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 area. In this new technique, both area and delay are considered. As the number of submachines is usually larger in the proposed technique, it can be expected that the delays reported here are better than those using the method in [3]. The data obtained could not be compared with other data 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 and the other machine can be in symbolic, encoded or encoded & 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 Verification Procedures

The problem with the verification of FSM networks obtained from the decomposition of a single FSM is that often the state space of the decomposed machine becomes much larger than the state space of the prototype machine. This is because the state space of the decomposed machine is often represented as the product of the states of each submachine. The decomposed machine may have several states which are unused. This can be observed in most examples shown earlier in the decomposition section and among the examples of [13]. This makes the verification of a decomposed machine a very difficult task. Usual verification techniques may waste a large amount of time checking the larger state space. We use an efficient method to restrict our verification search to the state space of the prototype specification only, thus making the verification process 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 the prototype machine. 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.

For the verification of FSM networks obtained from the decomposition of a single FSM, we assume knowledge of the output distribution, i.e. the partitioning of the primary outputs of the prototype machine among the submachines. Below we present an example to explain what we mean by output distribution. All the outputs of the prototype machine need not be assigned to one submachine. To verify the output of the decomposed machine it is necessary to reconstruct the output vector correctly. The output distribution information is provided in a decomposition file with the output labels (see section 7). The algorithm uses a simple state mapping table to keep the correspondence of the states of the prototype machine to the states of the submachines. These two pieces of information together allow the algorithm to verify the output as well as the next states of the decomposed machine. The state map table may be derived from information given in the decomposition file as described in section 5.2, or it can be generated by the method described in section 5.4, if the decomposition

Table 3: Example prototype machine

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

information is not provided.

Example 4: Suppose we have a prototype machine M with 5 primary outputs - $Q_1, \ldots, Q_5$ and 4 states $S = (A : B : C : D)$. Suppose this machine is decomposed into 2 submachines, M_1 and M_2, such that each submachine has 2 states. If output Q_1, Q_3 and Q_4 are in submachine M_1 and outputs Q_2 and Q_5 are in submachine M_2 then the output distribution is $O = (Q_1, Q_3, Q_4 : Q_2, Q_5)$. ■

5.2 Verification with Decomposition Information

The verification tool can be used at various stages of the synthesis process. The submachine files are treated differently at each of these stages. We can get two situations in the verification of FSM networks. One, in which the network has been obtained from the decomposition of a single FSM with the decomposition information available. By decomposition information we mean information on how the states of the prototype machine were partitioned to create the states of the decomposed machine. The other is when we have a network of FSMs obtained from a high level description but we don't have the mapping of the submachine states to the prototype machine states, i.e. we don't have the decomposition information. Such a network may be obtained from a VHDL specification, where each FSM corresponds to a VHDL process. Our verification algorithm is general enough to work in both situations. Thus the tool can be used to verify networks of FSMs obtained from high level specification using description languages such as VHDL, where the decomposition is implicit in the specification.

Example 5: Let us first explain the meaning of decomposition information with an example. Suppose we have a prototype machine M with 4 states, $S = (A, B, C, D)$ as shown in Table 3. Assume that it is partitioned into two submachines, M_1 and M_2 with the following partitions: $\pi_1 = \{A, B : C, D\} = (\alpha : \beta)$ and $\pi_2 = \{A, C : B, D\} = (\gamma : \delta)$. Thus we know exactly which state of the prototype machine maps to which state of the decomposed machine. For example, state B of the prototype machine maps to state α of submachine M_1 and to state δ of submachine M_2. Thus when the prototype machine is in state B the decomposed machine should be in state (α, δ). Information contained in partitions π_1 and π_2 is referred to as the decomposition information. ■

To verify a system when the decomposition 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 decomposition information is known, the next state of each submachine corresponding to the next state of the prototype machine is readily determined. Thus the next state is compared and verified to be the same in both machines. The outputs of the prototype machine are also verified with the outputs of the decomposed machine. The decomposed machine is said to be equivalent to the prototype specification if every transition edge of the graph is verified for the output as well as the next state. The details of the graph traversal technique are explained in the next section.

5.3 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 procedure to build the state map table is given in the next section. The state map table has the correspondence information between the states of the prototype machine and the states of the submachines.

The verification algorithm begins with both the prototype and the decomposed machines assumed to be 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. Otherwise, the next step is to enumerate 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 will be the present state for the next iteration of the algorithm. The input vector is then applied as the simulation input to the decomposed machine and the next state and outputs are obtained.

We check if the output of the decomposed machine implies the output of the prototype machine. If the output checking is successful then we check if the next state of the decomposed machine matches the mapping of the prototype machine according to the state map table. The map table may contain more than one corresponding states of the decomposed machine for every state of the prototype machine. Thus in general it is a one to many mapping. This takes care of equivalent states in the decomposed machine. The operations outlined above are 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. The algorithm backtracks to a previous state at this stage. 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. The output distribution information is used for this purpose.

The main steps of the algorithm are shown in Table 4. The procedure **traverse_DFS** performs the traversal of the STG as described above. The procedure verifies the two ma-

Table 4: Main verification procedure using enumeration

```
/* M_1 and M_2 are the machines to be verified */
/* M_1 being the prototype machine */

Procedure Verify(M_1, M_2)
   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
```

chines by traversing every edge of the STG only once. This is in contrast to some other enumeration-simulation approaches where every path (rather than edge) of the STG has to be traversed [10]. Thus we can see that the complexity of our algorithm is in $\Theta(E+N)$, the complexity of 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 [10] is in $\Omega(E+N)$ Our approach is more efficient because we verify the output and the next state of each transition of the machine under test, while the approach in [10] 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. This is the main reason for our method to be more efficient than other existing methods.

Example 6: Suppose we have a prototype machine M with 4 states, $S = (A, B, C, D)$. Assume that it is partitioned into two submachines, M_1 and M_2 with the following partitions: $\pi_1 = \{A, B : C, D\} = (\alpha : \beta)$ and $\pi_2 = \{A, C : B, D\} = (\gamma : \delta)$ as in example 5. Fig 5 shows the STG of the prototype machine on the left and the STG of the submachines on the right. Suppose the reset state of the prototype machine is state A and for the decomposed machine it is state (α, γ) (notice that $\alpha.\gamma = (A, B).(A, C) = A$).

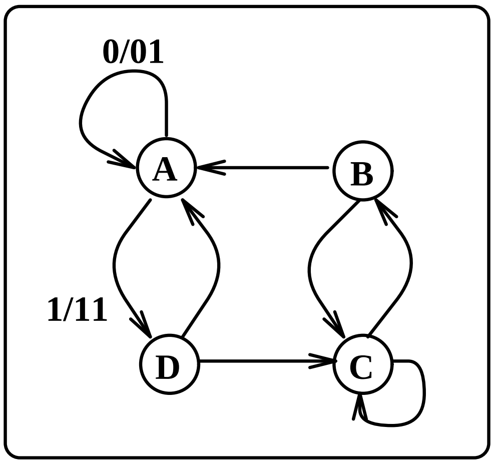
Our enumeration-simulation algorithm will begin from the reset state of both machines. Thus the prototype machine will be in state A while the decomposed machine will be in state (α, γ). Suppose we first traverse the edge corresponding to the loop on state A in the STG of the prototype machine. The input vector corresponding to this edge is 0. If we apply an input 0 to the decomposed machine, it loops back to state α in M_1 and γ in M_2, i.e. to state (α, γ). We verify the output bits to be 01 for both machines. Hence, we have verified this transition edge.

Now we pick the next untraversed fanout edge from state A. This corresponds to the edge $A \to D$. The corresponding input vector is 1. We apply an input of 1 to the decomposed machine. This takes submachine M_1 to state β and submachine M_2 to state δ. The output generated is 11 which verifies correctly with the prototype output. We now verify the next state. State $\beta = C, D$ in M_1 contains state D and state $\delta = B, D$ in M_2 also contains state D (this information would be contained in the state map table). We can clearly see that $\beta.\delta = D$. Hence the next state of the decomposed machine is also correct. So we continue to the next edge.

Suppose we now pick the edge going from D to A and we verify that also. Now, we are back at state A and we have checked all fanout edges of A. So we backtrack to D and pick another fanout edge of D. We continue verifying the transition edges similarly till we have covered all edges of the STG. A possible path to traverse all the edges is $(A \to A \to D \to A - D \to C \to C \to B \to C - B \to A)$. ■

5.4 Verification Without Decomposition Information (Generation of State Map Table)

A key feature of our system is the generation of the decomposition information for the decomposed machine. This allows the system to verify FSM networks obtained from high level specifications. If the decomposition information (partition of states) is not known the system first attempts to find a mapping of the prototype states to the submachine states thus generating the decomposition information. The reset states of both machines form the first entry of the state map table. This is the starting point of the algorithm to generate the decomposition information by building the state map table. The prototype and the decomposed machines are taken to their reset state. We then pick an edge of the state transition graph of the prototype machine and apply the corresponding input to the decomposed machine. The next state of the decomposed machine, which is the product of the corresponding next states of all its submachines, gives the mapping of the next state of

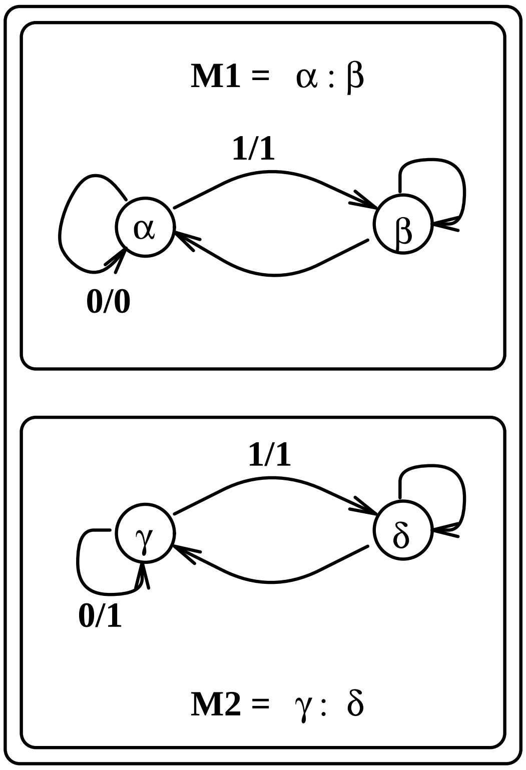


(a)

State Map Table

A	α, γ
B	α, δ
C	β, γ
D	β, δ

(b)



(c)

A -> A -> D -> A - D -> C -> C -> B -> C - B -> A

Figure 5: Example showing possible traversal path

the prototype machine and is added to the look-up table. By traversing different edges of the STG, the look-up table is completed for every state of the prototype machine.

It is often possible to obtain equivalent states in the decomposed machine since the state space of the decomposed machine is much larger than the state space of the prototype machine. The above mentioned approach to generate the state mapping of the two machines may give a wrong result when the decomposed machine contains equivalent states. In this case we check for equivalence of the states of the decomposed machine whenever necessary. If we arrive at a state of the decomposed machine which does not match the state in our look-up table, we check for the equivalence of the states before declaring that the two machines are not equivalent. If the two states are equivalent then the additional mapping is added to the state map table. The procedure is illustrated by the example below.

Example 7: Figure 6(a) shows the STG of a prototype machine with 4 states (same as in example 6) while Figure 6(c) shows the STG of the submachines of its decomposed machine. Submachine M_1 has 3 states (corresponding to states $\alpha = (A, B)$, $\beta = (C, D)$ and $\phi = (A, B)$ of the prototype). Suppose the reset state of the prototype machine is state A and for the decomposed machine it is state (α, γ). Figure 6(b) shows part of the state map table. The first column in the table has the name of a state of the prototype machine. The second column contains the names of the states of the submachine which correspond to that state of the prototype machine. The second column will have as many elements in it as the number of submachines with one state name for each submachine. Usually we will have only two columns in this table. However, in case of equivalent states in any of the submachines we create additional columns depending on the number of equivalent states we have for each state.

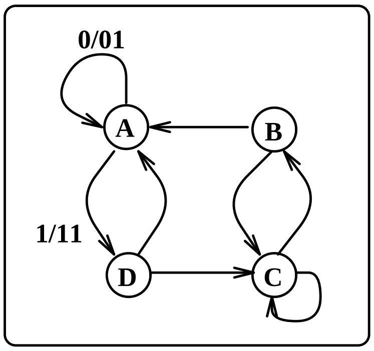
Our procedure to build the state map table begins with both machines in their reset states, i.e. state A for the prototype machine and state (α, γ) for the decomposed machine. Suppose, we now apply an input of 0 to both machines. The prototype machine stays in state A while the decomposed machine goes to state (ϕ, γ). This generates a conflict in our state map table in the sense that the states of the submachine do not agree with the state names already existing 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 error at this stage, the algorithm will check for the equivalence of states α and ϕ in submachine M_1. Since the two states happen to be equivalent, we add one more entry (add a column in the table) in the state map table for state A of the prototype machine as shown in the partial map table in Fig. 6(b). The algorithm will continue similarly to complete the state map table till we have an entry for all the states of the prototype machine. ■

Once the state map table is completed for all the states of the prototype machine we follow the same verification procedure as for the case when the decomposition information is known. During future simulations, if the prototype machine arrives at a given state we expect the decomposed machine to arrive at the same product of its submachine states as given in the state map table. If the prototype state maps to several equivalent states of the decomposed machine, then after each simulation step the state of the prototype machine should verify with at least one of those equivalent states. If this is not so, we declare the two machines to be different. As above, during the verification stage, the outputs are also compared.

5.5 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. At this stage we would like to verify the encoding process. The problem of verifying an encoded machine is very similar to verifying an unencoded machine or a machine at the symbolic level.

An encoded but non-optimized machine can be considered as a machine at the symbolic level since we can consider the codes of the states to be the symbols of the states. Since there are no don't cares, every state has a unique symbol represented by its binary code, the above mentioned verification method can be easily used to verify the encoded decomposed machine.

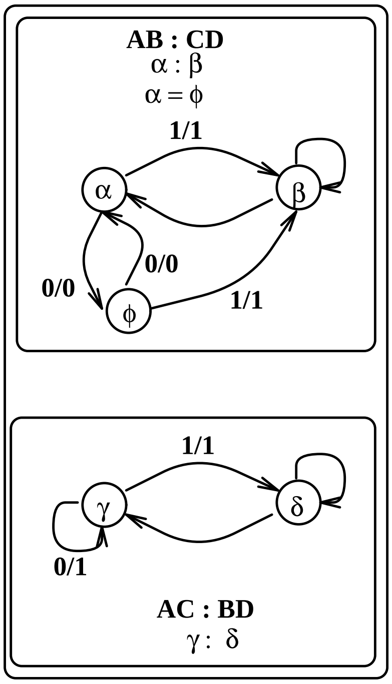


(A)

Part of Map_Table

A	α, γ	ϕ, γ
D	β, δ	

(B)



(C)

Figure 6: Submachine with equivalent states

5.6 Verification of Optimized Machine

The verification of the optimized machine (network of FSMs) implies verification of the synthesis/optimization process. We verify the binary covers of each optimized submachine with a corresponding cover generated from the prototype machine. The encoding and decomposition information is used to generate the unoptimized cover for each submachine from the prototype machine. This cover is then compared with the optimized cover of that submachine.

The following procedure is used to generate from the prototype machine an unoptimized cover $C_p^f(M)$ for each submachine M. This will then be compared to the optimized covers of the corresponding submachine. For each transition edge in the STG of the prototype a corresponding cube is added to the cover of the corresponding submachine. The input field of the cube is the concatenation of the primary input and the binary code of the present state of the submachine. Similarly, the output field of the cube is the concatenation of the next state code and the output bits present in that submachine. The output distribution information is used to obtain the correct output bits. This is necessary because the cover of a submachine is going to contain the corresponding output bits only. The coding of the states is obtained from the submachine files. This procedure generates an unoptimized cover from the prototype machine for each submachine. The on-set and off-set of the optimized and unoptimized covers thus obtained are compared to verify the equivalence of each submachine. If all submachines are found to be equivalent then the two machines are declared equivalent.

Definition 5.1 *Let the optimized on-set and off-set covers of submachine M be $C_o^f(M)$ and $C_o^r(M)$ respectively. The unoptimized on-set and off-set of the corresponding covers generated from the prototype machine be $C_p^f(M)$ and $C_p^r(M)$ respectively. $C_p^f(M)$ is the unoptimized cover obtained from the prototype machine corresponding to submachine M as described above.*

The decomposed machine is said to be equivalent to the prototype machine, if for each submachine M of the prototype machine, $(C_p^f(M) \subseteq C_o^f(M))$ and $(C_p^r(M) \subseteq C_o^r(M))$. At this stage we are concerned with the verification of the minimization procedure and we assume that the decomposition is legal. To check the equivalence of the two machines we check the equivalence of the unoptimized and optimized covers of the submachines. The condition $[C_p^f(M) \subseteq C_o^f(M)]$ implies that every time the prototype machine has a 1 at its output the submachine also has a 1 at its output. The condition $[C_p^r(M) \subseteq C_o^r(M)]$ implies that every time the prototype machine has a 0 at its output, the submachine output is also a 0. The satisfaction of both these conditions implies that the submachine is equivalent to its corresponding unoptimized submachine obtained from the prototype. Clearly, the on-set of the optimized submachine must cover the on-set obtained from the prototype without intersecting the off-set, but the two may not have identical on-sets. This is because the two covers may have different don't-care sets. If each submachine cover is equivalent to its corresponding unoptimized prototype cover, then the decomposed machine is equivalent to the prototype machine. Since 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, the above condition is sufficient. Our program uses Espresso routines to compare the submachine covers.

6 Verification Tool

The various algorithms described above have been implemented into a program called *Ver-Sim*. Apart from verifying the decomposed machine with respect to the prototype machine, the program has several other options which may help a designer in the synthesis of an FSM network. Various other features and tool options are described below.

6.1 Errors Checked

The verification tool checks for basically three types of errors and displays the error message with the appropriate label. 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 decomposition errors. The verification tool reads the decomposition file and checks for errors in the decomposition itself. Some of these errors may be detected while verifying the submachine files. The tool continues to verify the rest of the file even after locating a decomposition error and reports all other errors present. Errors detected 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 are transition errors. The tool starts verifying the submachine files. Transitions to wrong next states and 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. This is important because the prototype machine and the decomposed machine are said to be equivalent only if every transition present in the prototype is also present in the decomposed machine.

6.2 Simulation Tool

The simulation option of the system is invoked by the $-s$ option of the program *VerSim*. Simulation can be done interactively or in a batch mode through a simulation file. If a simulation input file is provided using the $-i$ option then it runs in the batch mode, otherwise it defaults to the interactive mode. In the interactive mode the system prompts the user for an input. It generates the proper output and next state for every input given taking the machine under simulation through its state transition graph. The simulation can be controlled using various control commands. All control commands begin with a dot. The machine can be brought back to its reset state by the *.reset* command. The current state and outputs of the submachines under simulation can be viewed by typing *.print*. A record of all the simulation steps can be made by the log file option which maintains a log of all the inputs and outputs. This file can be used for later simulations.

In the batch mode the simulation input file should contain a list of input vectors which have to be provided to the machine. All the control commands can also be given in the simulation input file.

6.3 Submachine Generation Tool

The generation tool option is invoked by the $-g$ option on the command line. This creates the submachine files from a given prototype file and its decomposition. The format of the prototype file and the decomposition file are the same as for the verification option described above. The submachine files can be used by the tool for verification and simulation. This option can be used for experiments with different decompositions. Given a decomposition it can very quickly check its legality and generate the submachine files.

Table 5: Command line options

Option	Description
-d	Verify with known decomposition
-u	Verify with unknown decomposition
-c	Verify an optimized machine
-g	Generate state transition table
-s	Simulate system
-i [file]	Simulation input file; optional
-l [file]	Output log file; optional (can be used during simulation)

7 System Overview

The system has been implemented in C and runs on a Unix operating system using simple text files for input and output. All the verification errors are printed on the standard output with detailed textual references to the possible source of the error. The program can take several arguments as its input depending on the option which the user wishes to use. Different command line options allow the tool to be used during different stages of the synthesis process. The various command line options are given in Table 5.

The system reads the submachine and the network decomposition information and combines the two for simulation and verification. The prototype specification, the submachines and the network configuration are described in different files as described below.

Prototype File: This is the main file which contains the description of the prototype machine which will be used for the verification of the decomposed submachines. The format of the prototype file is the Kiss-style format.

Decomposition File: This file contains various statistics about the decomposed machine network, like the number of submachines in the system. It also contains the names of the outputs which are present in each submachine and may also contain the names of the states of the prototype machine which are combined in each submachine. Various other data about the network can be optionally put in this file. An example decomposition file is shown in Table 6.

Submachine Files: There should be as many submachine files as the number of submachines in the network. These files contain the STTs of the submachines in symbolic form obtained from decomposition or in binary form after the encoding of the states. The symbolic form of the submachine file format is very similar to that of the prototype file. The only difference is in the control lines because the Kiss-style format does not support multiple symbolic inputs. The submachine file has a control line to define multiple symbolic inputs as used in Espresso-mv. An example of a submachine file is shown in Table 7. It has five variables: a single binary variable and 4 multiple-valued variables, including the output. The

Table 6: Example decomposition file

.m 2	# Number of submachines
.n test1 test2	# Names of submachines
.sn test1 2	# Number of states in submachine
.ss test1 A,D,F : B,C, E,G	# Decomposition
.in test1 1	# Number of primary inputs
.on test1 2	# Number of primary outputs
.os test1 0 1	# Names of outputs
.sn test2 4	
.ss test2 A,B : D,G : F,E : C	
.in test2 1	# This line is optional
.on test2 1	
.os test2 2	

first multiple-valued variable has 4 values while the others have 2 values.

8 Results and Conclusion

The program $VerSim$ has been tested on several MCNC examples and some industrial circuits. All circuits were decomposed using the decomposition algorithm described earlier. The program could successfully verify all the examples shown in Table 8. The program was tested using all the verification options described earlier. The table shows the statistics of the examples tried and the run time for verification with the '$-d$' and '$-u$' option. The '$-d$' option is for the case when the decomposition information is known while the '$-u$' option is used when the decomposition information is not known. The first 4 columns in the table give the number of inputs, states, transition edges and outputs in the example. The column labeled " # subm" gives the number of submachines of the decomposed machine. The last two columns give the time for verifying the system using the '$-d$' and '$-u$' option, respectively. The time given in the table is time in seconds while running the program on a DECstation 5000/125 machine.

We have shown the potential and efficiency of our program to verify decomposed machines with their prototype specifications. We have verified in reasonable time machines with up to 1800 edges and up to 121 states. The 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 method can be used at various stages of the synthesis process to check the correctness of each transformation step. A modified version of the algorithm is presented for the case when the decomposition information is not available. In all cases the modified version took longer time to verify the machines because of the

Table 7: Example submachine file

```
.mv 5 1 4 2 2 2
0 b1 a1 a1 00
1 b1 a1 a1 00
0 b1 a2 a2 00
1 b1 a2 a2 00
0 b4 a2 a2 00
1 b4 a2 a2 00
0 b2 a1 a1 00
1 b2 a1 a1 10
0 b3 a2 a1 10
1 b3 a2 a2 10
0 b3 a1 a1 01
1 b3 a1 a2 01
0 b2 a2 a2 00
1 b2 a2 a1 10
.e
```

Table 8: Results

Name	inputs	states	trans edges	outputs	# subm	$time^d$	$time^u$
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

time spent in building the state map table. The modified version can be used to verify FSM networks obtained from a VHDL description also. Since the VLSI circuits of today are generally very large and complex systems, efficient verification tools can serve as useful development aids.

Acknowledgments

The authors wish to thank Maya Yajnik for providing in-depth analysis of the decomposition method presented and for developing the C program.

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] 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.

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

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

[7] 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.

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

[9] 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.

[10] 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.

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

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

[13] 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.

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

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

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

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

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

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

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

[21] 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.

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

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