VLSI Design
1994, Vol. 2, No. 2, pp. 105–116
Reprints available directly from the publisher
Photocopying permitted by license only

Pioneer: A New Tool for Coding of Multi-Level Finite State Machines Based on Evolution Programming

S. MUDDAPPA
Cirrus Logic, Fremont, CA 94538

R.Z. MAKKI and Z. MICHALEWICZ
University of North Carolina at Charlotte, Charlotte, NC 28223

S. ISUKAPALLI
Alliance Semiconductor, San Jose, CA 95112

In this paper we present a new tool for the encoding of multi-level finite state machines based on the concept of evolution programming. Evolution programs are stochastic adaptive algorithms, based on the paradigm of genetic algorithms whose search methods model some natural phenomenon: genetic inheritance and Darwinian strife for survival. Crossover and mutation rates were tailored to the state assignment problem experimentally. We present results over a wide range of MCNC benchmarks which demonstrate the effectiveness of the new tool. The results show that evolution programs can be effectively applied to state assignment.

Key Words: *Evolution programming; State assignment; Finite state machines; Crossover; Mutation*

1 INTRODUCTION

Automated logic synthesis of Finite State Machines (FSM's) typically begins with a high level description of the FSM and ends with an appropriate gate level netlist. Beginning with a state transition table the process of logic synthesis can be subdivided into: state assignment, two-level minimization, multi-level minimization, and technology mapping. The state assignment problem has been the object of extensive research as it is a key step in the synthesis process influencing the resulting silicon area, the speed of the circuit, and the testability of the circuit [16, 23, 24]. Most of the early work in automatic state assignment has been directed at the minimization of the number of product terms in a two-level sum-of-products form of the combinational logic. Examples of tools aimed at two level minimization include KISS [7] and NOVA [25]. However, in recent years state assignment tools have emerged which target multi-level minimization. Examples of tools targeting multi-level minimization include JEDI [15], MUSTANG [22], MUSE [27] and MARS [24]. However most of these tools suffer from two main drawbacks. First is the limited accuracy of the cost function used to estimate the goodness of the solution including the weighing mechanism used for weighing the various constraints. The second is the limited efficiency of the graph embedding algorithms. Generally, these algorithms are based on hill climbing techniques which represent local searches of the solution space; hence they fail to adequately explore new neighborhoods.

In this paper we address the embedding problem by investigating the feasibility of using an evolution programming technique for the state assignment problem and tuning of its parameters (frequencies of crossover and mutation operators) in order to obtain a good convergence rate. Evolution programs (EP) [18] are stochastic adaptive algorithms, based on the paradigm of genetic algorithms [5, 8, 12] whose search methods model some natural phenomenon: genetic inheritance and Darwinian strife for survival. They can be used for finding near optimal solutions to various large scale optimization problems, many of which have been proved NP-hard.

This paper discusses the development of a new evolution programming tool, Pioneer, for the state

assignment problem. The advantage of Pioneer is that it exploits both local neighborhoods and new regions of the search space to find a solution. Pioneer is incorporated with three common cost functions as user-specified options.

This paper is organized as follows. Section 2 defines the state assignment problem and summarizes prior work on state assignment. In Section 3 we provide an introduction to genetic algorithms and their generalization, evolution programs. Section 4 describes the development of Pioneer. Section 5 presents experimental results and Section 6 presents conclusions and also gives directions for future work.

2 STATE ASSIGNMENT

2.1 Problem Formulation

An FSM is composed of a combinational network that computes the next-state and output logic, and a set of flip-flops, whose binary values within a given clock period represent the state of the FSM. The state assignment, which involves developing a unique binary code representation for each individual state, plays a central and crucial role in logic synthesis. It has a direct and critical influence on circuit silicon area, circuit speed, and circuit testability [23, 24]. The parameters to be considered include:

1. the length of the binary code (number of bits in the code), and
2. the composition of the code (which code is assigned to which state).

The optimization problem relative to state assignment traditionally involves optimizing the above parameters in order to achieve one or more of the following objectives.

1. Minimum number of literals (a literal is a state variable or its complement), representing the amount of logic in the combinational network of the FSM, as a measure of silicon area occupied by the resulting FSM circuit. This can target either two-level FSM implementations such as PLA's or multi-level implementations.
2. FSM testability. This usually involves maximizing controllability and observability parameters. For example it has been found that a one-hot (distance 2) assignment, where the code length is equal to the number of states and the code for state i consists of a string of 0's and only a single 1 placed the ith position, results in easily testable FSM's [16].

In this paper, we address the problem of state assignment in order to minimize circuit area for the case of multi-level minimization. Our goal is as follows: for a given FSM, find a set of state code representations for the states of the FSM that minimizes the cost function C(NSL/OL, F), where NSL/OL represents the number of literals in the next-state and output logic block and F represents the number of flip-flops. It is very difficult to find an expression for C that accurately estimates the cost of an assignment without performing actual minimization (a process which consumes too much time to be incorporated into a state assignment tool). As stated above, cost functions have been derived to provide an early estimate of the results of minimization (see Section 5). These cost functions take into account encoding relationships among the different states which are used by FSM minimization tools without actually performing minimization.

2.2 Previous Work

The State Assignment Problem which belongs to the class of NP-hard (O(n!)) problems, has been the object of extensive research since the 1960's [1, 9, 10, 13, 14]. In classical approaches to this problem, Armstrong developed a method capable of coding large machines, based on a graph interpretation of the problem. He transformed the state assignment problem into a graph embedding problem. Dolotta and McCluskey [17] introduced the concept of codable columns as a method for estimating the complexity of the combinational component of a finite state machine. This method is not effective for large FSM's.

More recently state assignment tools have been developed so as to target a particular implementation. For example tools targeting two-level implementations include KISS [7] and NOVA [25], and tools targeting multi-level implementations include MUSTANG [22], JEDI [15] and MUSE [27].

For two level implementation, both KISS and NOVA generate constraint groups which are subsets of states. The states in a constraint group generate the same next state and the same outputs under the same input pattern. In order to reduce the number of product terms in a PLA implementation, a heuristic encoding procedure tries to find a state assignment which satisfies these group constraints. A group constraint is satisfied by assigning the group to a minimum dimension boolean subspace. States be-

longing to groups outside those in a given group constraint, GC, cannot be given assignments which intersect the boolean subspace associated with GC. An encoding length longer than the *minimum encoding length* ($\lceil \log_2(n) \rceil$, where n is the number of states) may be required to satisfy the constraints. In practice, most large FSM's cannot be synthesized as a single PLA for performance reasons—multi-level logic implementations are generally used for smaller delays.

Multi-level targeted encoding algorithms attempt to take advantage of multi-level implementations and to make use of multi-level logic optimization tools such as BOLD [11] and MIS [3]. The MUSTANG algorithm [22] is primarily based on algebraic techniques applied to a two-level representation of an FSM. The heuristic encoding algorithm tries to maximize the number and size of common subexpressions so that multi-level logic optimization programs like MIS and BOLD can factor out the common subexpressions and create an optimized multi-level implementation of a state machine. The JEDI [15] algorithm, takes a similar approach, but addresses a more general symbolic encoding problem.

Su [24] developed a KISS-like state assignment program called MARS which varies the length of the binary code for finding an optimum assignment. The length of the binary code is bound by the above *minimum encoding length*. However the *minimum encoding length* does not always give the best results. The optimum length of the binary code is sometimes greater than the *minimum encoding length* but less than or equal to the number of states.

The MUSE algorithm [27] uses a multi-level structure derived by performing multi-level optimization on a 2-level structure obtained from 1-hot encoding. From the multi-level representation, the algorithm evaluates the affinities of each unique pair of states, in terms of how the hamming distance between the codes assigned would affect the overall network size after multi-level synthesis.

3 GENETIC ALGORITHMS AND EVOLUTION PROGRAMS

In this Section we introduce genetic algorithms, present their theoretical foundations, and describe their applicability. Further, we discuss evolution programs.

3.1 Basic Concepts

Genetic algorithms (GAs) represent a class of adaptive algorithms whose search methods are based on simulation of natural genetics. They belong to the class of probabilistic algorithms; yet, they are very different from random algorithms as they combine the elements of directed and stochastic search.

In general, a GA performs a multi-directional search by maintaining a population of potential solutions. This population undergoes a simulated evolution: at each generation the relatively 'good' solutions reproduce, while the relatively 'bad' solutions die. To distinguish between different solutions, an evaluation function is used which plays the role of an environment.

The structure of a simple GA is shown in Figure 1. During iteration t, the GA maintains a population of potential solutions (called chromosomes following the natural terminology), $P(t) = \{x_t^1, \ldots, x_t^n\}$. Each solution x_t^i is evaluated to give some measure of its 'fitness.' Then, a new population (iteration $t + 1$) is formed by selecting the more fitted individuals. Some members of this new population undergo reproduction by means of crossover and mutation, to form new solutions.

Crossover combines the features of two parent chromosomes to form two similar offsprings by swapping corresponding segments of parents. For example, if the parents are represented by five-dimensional vectors (a1, b1, c1, d1, e1) and (a2, b2, c2, d2, e2) (with each element called a gene), then crossing the chromosomes after the second gene would produce the offspring (a1, b1, c2, d2, e2) and (a2, b2, c1, d1, e1). The intuition behind the applicability

```
Procedure genetic algorithm
begin
    t = 0
    initialize P(t)
    evaluate P(t)
    while (not termination-condition) do
    begin
        t = t +1
        select P(t) from P(t -1)
        recombine P(t)
        evaluate P(t)
    end
end
```

FIGURE 1 A simple Genetic Algorithm.

of the crossover operator is information exchange between different potential solutions.

Mutation arbitrarily alters one or more genes of a selected chromosome, by a random change with a probability equal to the mutation rate. The intuition behind the mutation operator is the introduction of some extra variability into the population.

GAs require five components:

- a genetic representation for potential solutions to the problem,
- a way to create an initial population of potential solutions,
- an evaluation function that plays the role of the environment, rating solutions in terms of their "fitness,"
- genetic operators that alter the composition of children during reproduction, and
- values for various parameters (population size, probabilities of applying genetic operators, etc.)

3.2 Mathematical Foundation

The theoretical foundations of GAs rely on a binary string representation of solutions, and on the notion of a schema [12]—a template allowing exploration of similarities among chromosomes. A *schema* is built by introducing a *don't care* symbol (*) into the alphabet of genes—such *schema* represents all strings (a hyperplane, or subset of the search space) which match it on all positions other than '*'. For example, a schema H = '**101**' represents 16 strings: '0010100', '0010101', . . . , '1110111.' In a population of size n of chromosomes of length m, between 2^m and $2^m n$ different schemata may be represented; at least n^3 of them are processed in a useful manner: these are sampled at a (desirable) exponentially increasing rate, and are not disrupted by crossover and mutation. Holland [12] has called this property an *implicit parallelism,* as it is obtained without any extra memory or processing requirements.

There are two other important notions associated with the concept of the schema. First is the schema order, $o(H)$, which is the number of non don't care positions in the H: it defines the specialty of a schema. Second is the schema defining length, $l(H)$, which is the distance between the first and the last non-don't care symbols of H. For example, a schema H = '**101**' has $o(H) = 3$ and $l(H) = 2$.

Assuming that the selective probability is proportional to fitness, and independent probabilities, p_c and p_m, for crossover and mutation, respectively, we can derive the following growth equation [8]:

$$m(H, t+1) \geq m(H, t) \times \frac{f(H, t)}{\bar{f}(t)} \times \left[1 - p_c \frac{l(H)}{l-1} p_m o(H)\right]$$

where $m(H, t)$ is the expected number of chromosomes satisfying schema H at time t, $f(H, t)$ is the average fitness of schema H at time t, and $\bar{f}(t)$ is the average fitness of the population (at time t). The growth equation above shows that the selection increases the sampling rate of the above-average schemata, and that this change is exponential. The sampling itself does not introduce new schemata (not represented in the initial $t = 0$ sampling). This is exactly why the cross-over operator is introduced, to enable structured yet random information exchange. Additionally, the mutation operator introduces greater variability into the population. In short, GA seeks near-optimal performance through the juxtaposition of short, low-order, high-performance schemata, called the building blocks.

Recently a notion of so-called evolution programs (EP's) was proposed [18]. Roughly speaking, an EP is a genetic algorithm enhanced by problem specific knowledge; this knowledge is incorporated in appropriate data structures and problem specific operators.

The idea of incorporating a problem specific knowledge in genetic algorithms is not new and has been recognized for some time. In [6] Davis wrote:

"It has seemed true to me for some time that we cannot handle most real-world problems with binary representations and an operator set consisting only of binary crossover and binary mutation. One reason for this is that nearly every real-world domain has associated domain knowledge that is of use when one is considering a transformation of a solution in the domain [. . .] I believe that genetic algorithms are the appropriate algorithms to use in a great many real-world applications. I also believe that one should incorporate real-world knowledge in one's algorithm by adding it to one's decoder or by expanding one's operator set."

However, the concept of EP's [18] is different from the previously proposed ones. It is based entirely on the idea of genetic algorithms; the difference is that we allow any data structure (i.e., chromosome representation) suitable for a problem together with any set of "genetic" operators, whereas classical GAs use fixed-length binary strings for its individuals and two operators: binary mutation and binary crossover. In

other words, the structure of an evolution program is the same as the structure of a GA (Figure 1) and the differences are hidden on the lower level. Each chromosome need not be represented by a bit-string and the recombination process includes other "genetic" operators appropriate for the given structure and the given problem.

Already some researchers have explored the use of other representations as ordered lists (for bin-packing), embedded lists (for factory scheduling problems), variable-element lists (for semiconductor layout). During the last ten years, various application-specific variations on the genetic algorithm were reported; these variations include variable length strings (including strings whose elements were if—then—else rules [21]), richer structures than binary strings (for example, matrices [26], and experiments with modified genetic operators to meet the needs of particular applications [19]. In [20] there is a description of a genetic algorithm which uses back-propagation (a neural network training technique) as an operator, together with mutation and crossover that were tailored to the neural network domain. Davis and Coombs [4] described a genetic algorithm that carried out one stage in the process of designing packet-switching communication network; the representation used was not binary and five "genetic" operators (knowledge based, statistical, numerical) were used. These operators were quite different to binary mutation and crossover.

It seems that a "natural" representation of a potential solution for a given problem plus a family of applicable "genetic" operators might be quite useful in the approximation of solutions of many problems, and this nature-modeled approach (evolution programming) is a promising direction for problem solving in general. We will use this approach in building Pioneer: an evolution program for encoding finite state machines.

4 PIONEER

In this Section a description of the system Pioneer is presented. A new individual representation is presented along with variations of the crossover operators and mutation operators. An individual representation is a vector of integers. Two crossover operators and two mutation operators are described.

4.1 Genetic Representation of Solutions

Consider the state assignment for the state diagram shown in Figure 2. The traditional state assignment

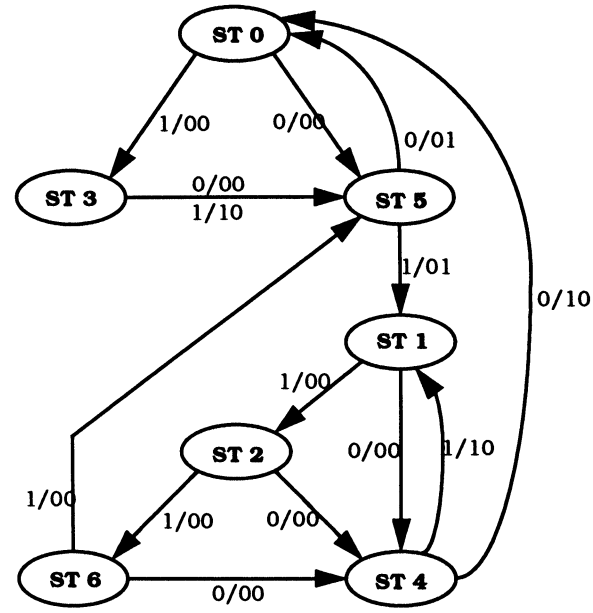

FIGURE 2 Example Finite State Machine.

would be as shown below. But in Pioneer, an integer set is used for representing the state assignment problem.

Traditional representation	Pioneer representation
State #0 = 011	State #0 = 3
State #1 = 000	State #1 = 0
State #2 = 111	State #2 = 7
State #3 = 100	State #3 = 4
State #4 = 101	State #4 = 5
State #5 = 001	State #5 = 1
State #6 = 010	State #6 = 2

This type of representation has a few advantages. If the chromosome were represented as a binary string, traditional crossover and mutation operators can often result in illegal offspring whereby two or more states are assigned the same code. In such a case, the state assignment tool must first change the code assigned to one or more state in order to ensure that each state has a unique code assignment. This is done by a so-called "repair algorithm." Repairing such an illegal chromosome, i.e., to force it into a feasible solution, would require conversion of binary strings into integers. On the other hand, integer representation does not require this step and thus the repair algorithm is simpler to implement, thus reducing the runtime of the process.

4.2 Initial Population

A population of individuals was initialized by generating random numbers between 0 to 2^{N_b}, where N_b

is the number of bits used for representing each state. Linked lists were used to increase the speed and to avoid the possibility of duplication of individuals. Another way of initializing the population would be by using the MUSTANG cost matrix to initialize half the population and the other half at random. This technique was not used in Pioneer as there is a possibility that the searching process would converge to a local minima. Starting the initial population at random would allow Pioneer to explore a larger area of the search space.

4.3 Genetic Operators

Two crossover and two mutation operators were used in Pioneer. Both crossover operators produce two offspring, whereas mutations produce single offspring. Some of these offspring go through a repair algorithm as they do not represent a legal solution, which require that no two states have the same code. Both crossover operators are applied with a given probability p_c, and mutation operators—with probability p_m.

The crossover operators combine the characteristics of both parents, and mutation operators introduce a small disturbance of a chromosome. We discuss all operators in turn.

Uniform Crossover

This crossover operator takes two parent individuals which are selected at random. As shown in the example below, a mask is formed at random. The next step is to swap the genes depending on the structure of the mask.

Example:

Mask: = [1 0 1 0 1 0 1] (formed at random)

Parent1: = [2 4 1 3 5 6 7]

Parent2: = [7 1 4 5 2 3 6]

Child1: = [7 4 4 3 2 6 7] unused states list: = [0 1 5]

Child2: = [2 1 1 5 2 6 6] unused states list: = [0 3 4 7]

The repair algorithm takes care of the repeated values. The repair algorithm randomly selects duplicate states and replaces them with states from the unused states list.

Child1: = [7 4 1 3 2 6 0]

Child2: = [2 1 0 5 3 6 7]

These two children replace the ones with the worst cost which is found using one of the cost functions.

One-point Crossover

This crossover operator also selects two parents at random. In this case, each of the selected individuals is split at a random point and recombined with the other. This results in two children having qualities of both the parents. An example of One-point crossover is shown below

Parent 1: = [2 4 1 7 | 0 5 3]

Parent 2: = [3 7 5 4 | 1 0 2]

After swapping the two parents, the resulting children are

Child1: = [2 4 1 7 1 0 2] unused states list = [3 5 6]

Child2: = [3 7 5 4 0 5 3] unused states list = [1 2 6]

Using the same repair algorithm, the resulting children are

Child1: = [2 4 1 7 6 0 5]

Child2: = [3 7 5 4 0 2 1]

Mutate1

After selecting an individual for mutation, two genes are selected at random from the individual and swapped.

Example: Parent = [2 4 0 7 1 3 5]

If the mutation operator selects two positions "4" and "6" (the leftmost position being 1), then the resulting chromosome would be

Child: = [2 4 0 3 1 7 5]

Mutate2

Two positions are selected at random as the positions to be mutated. The assignment at these positions is moved to the unused states list. Mutate2 then selects two numbers at random from these unused states list. This procedure tries to introduce more randomness into the system. This operator is applicable only if there is a non-empty list of unused states. In other words, this mutation is not used for tight state assignments problems where the number of states is 2^{N_b}.

Example:

Parent: = [4, 3, 1, 0, 15, 13, 10, 5, 14] unused states = [2, 6, 7, 8, 9, 11, 12]

If positions 3 and 6 are selected for mutation, then "1" and "13" are moved to the unused states list resulting in,

unused states = [2, 6, 7, 8, 9, 11, 12, 1, 13]

Two genes are selected at random from the unused states list.

The resulting Child: = [4, 3, 8, 0, 15, 12, 10, 5, 14]

unused states = [2, 6, 7, 9, 11, 1, 13]

4.4 Evaluation Functions

Evaluation functions or cost functions are necessary for any optimization process. The cost function provides a measure of the goodness of a solution. In the EP approach, each individual undergoes evaluation after each generation. The result of evaluation influences its chances of survival and reproduction. The cost of any assignment is computed by examining the encoding affinity between states which is a measure of the gain resulting from giving two states minimal distance assignments. The objective is to give minimal distance encodings for the groups of states possessing high encoding affinities. Built into Pioneer are three different cost functions which are provided as user definable options. In this Section, a brief description of the cost functions is given.

4.4.1 Present State Cost Function

In this cost function, the output information and the present state information of different states are considered to estimate the encoding affinity. The encoding affinity [27] between two pairs of states k and 1 is calculated using the relation,

$$P^p_{k,1} = \left(\sum_{i=1}^{N_o} p^o_{k,i} \times p^o_{l,i} + \sum_{i=1}^{N_s} \lambda_i \times p^N_{k,i} \times p^N_{l,i} \right)$$

where N_o is the number of outputs, N_s is the number of states, $p^o_{k,i}$ corresponds to the number of times state k asserts output i, $p^N_{k,i}$ corresponds to the number of transitions from present state k to next state i, and λ_i is the number of 1's in the encoding of state i. In tools such as MUSTANG and JEDI, λ_i is not known a priori and thus it must be estimated. However in Pioneer, each generation represents a legal set of assignments (i.e., each state has a unique assignment) and thus λ_i is known by simply counting the number of 1's in each assignment.

The total cost estimate after a state assignment is,

$$\text{TotalCost} = \sum_{k \neq l} (P^p_{k,l} \times (\text{HD}(k, l) - 1))$$

where, HD(k, l) is the hamming distance between the two states k and l.

4.4.2 Next State Cost Function

The next state cost function is based on the input and next state transitions of different states in the state transition table. Next states which are produced on similar input conditions and from similar sets of present states are given priority in the assignment of minimum distance codes.

Let A denote the input portion of the transition table T, A_i be the co-kernel of the algebraic expression associated with A, and $|A_i|$ be the number of literals in the co-kernel. The encoding affinity [27] between a state pair is given by,

$$P^n_{k,1} = \left(\sum_{i=1}^{|A|} n^I_{k,i} \times n^I_{l,i} + \sum_{i=1}^{N_s} \lambda_i \times n^P_{k,i} \times n^P_{l,i} \right)$$

where $n^I_{k,i}$ and $n^P_{k,i}$ correspond to the input field and present state field of T respectively. The first summation of the above cost function looks at state transitions controlled by the same input condition and the second term looks at states having common next states.

The total cost estimate after state assignment is,

$$\text{TotalCost} = \sum_{k \neq l} (P^n_{k,l} \times (\text{HD}(k, l) - 1))$$

4.4.3 Constraints as a Cost Function

Constraints as a cost function are formed using rules to identify states which are targeted to be given minimal-distance state assignments. The following four rules have classically been utilized and have been incorporated into Pioneer in order to pair states for minimal-distance code assignments:

Rule 1: States having a common next state.

Rule 2: Next states of a given state.

Rule 3: States having common unconditional outputs.

Rule 4: States having common conditional outputs based on identical input conditions.

The rules were modified so that the weight assigned to the constraint was based on the size of the common input field. The number of constraints satisfied was used to evaluate the goodness of an assignment.

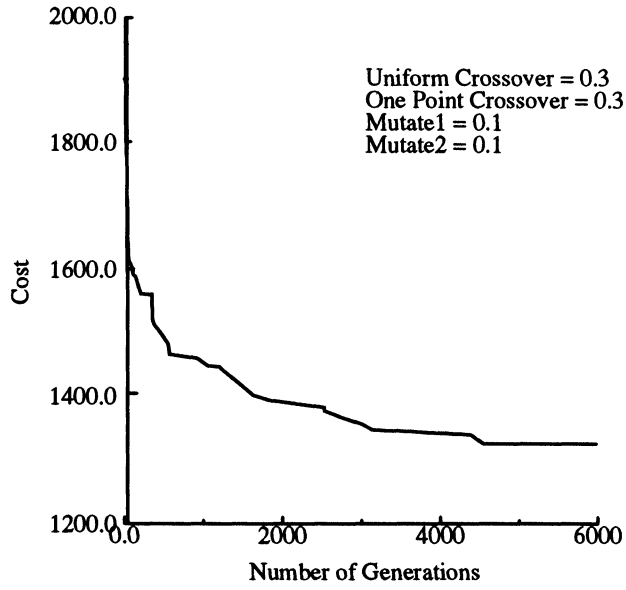

FIGURE 3 Cost "p" versus the number of generation (FSM sample cse).

5 EXPERIMENTAL RESULTS

Pioneer was used to perform experiments on MCNC (Microelectronics Center of North Carolina) FSM benchmarks. The experimental data is divided into two categories:

1. Convergence Speed (convergence towards good solutions), and
2. Effectiveness of Pioneer (comparison with other tools).

5.1 Convergence Speed

There are 3 options for running Pioneer: 'n' which refers to the next-state cost function option which is sought to be minimized, 'p' which refers to the present-state cost function option which is sought to be minimized, and 'c' which refers to the use of the above constraints as a cost function which is sought to be maximized. The convergence rate of the algorithm is depicted in Figures 3 to 5. The crossover and mutation rates are listed in the figures. The data is for the MCNC FSM sample "cse." Similar results were obtained for other FSM samples. The data in Figure 3 was obtained using the 'p' option, that of Figure 4 was obtained using the 'c' option, and that of Figure 5 was obtained using the 'n' option. From the graphs it is observed that the algorithm converges rapidly to good solutions. Increasing the crossover and mutation rates did not help in improving the performance of the algorithm. The mutation rates were found to be (via experimentation) much higher than those of classical genetic algorithm applications.

5.2 Effectiveness of Pioneer

Table II shows the effectiveness of Pioneer for 20 of the MCNC FSM benchmarks whose statistics appear

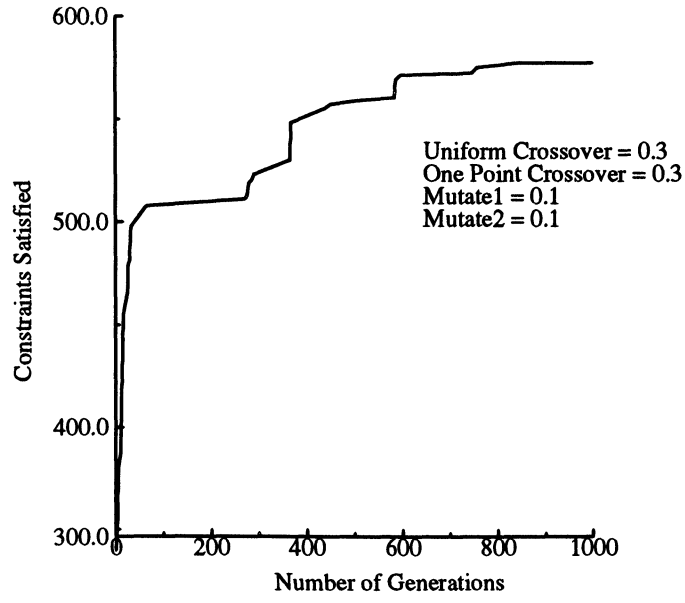

FIGURE 4 Cost "c" versus the number of generation (FSM sample cse).

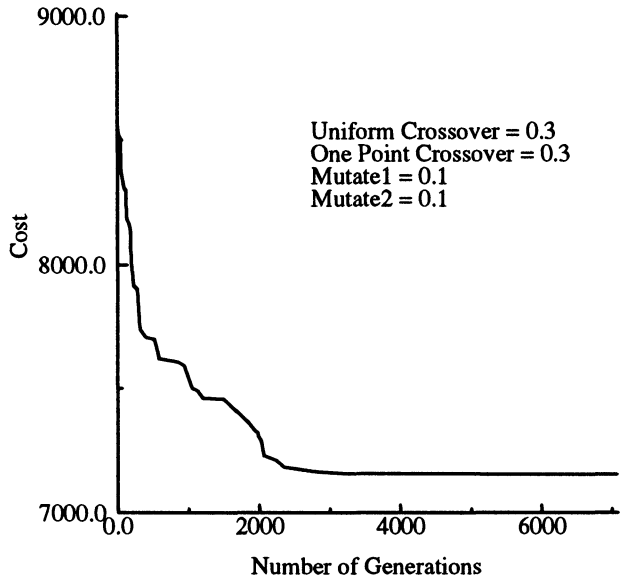

FIGURE 5 Cost "*n*" versus the number of generation (FSM sample cse).

in Table I. The "number of constraints satisfied" option produced the best results. The minimization tool ESPRESSO [2] was used to perform two level minimization and MIS-II [3] was used to perform multi-level minimization. The table depicts the actual literal count obtained from MIS-II in each case. The results were compared with those of MUSTANG [22] using both the '*p*' and '*n*' options which refer to the 'present-state' and 'next-state' cost function options respectively.

The Pioneer data was obtained by running Pioneer for an average of 1000 generations. Table II shows Pioneer to produce effective results. The option '*c*' produced the best results. The data in the row labelled "TOTAL" is simply the summation of all the literals in each column of the table. Due to the random nature of the Pioneer, this data is especially important because it gives a better idea of the overall performance of Pioneer for many runs of the program. Comparing the total number of literals of Pioneer to that of MUSTANG, we find that Pioneer's performance is very acceptable and produces superior results.

As in many genetic-based algorithms, the runtime of Pioneer is slow. For example, the '*n*' and '*p*' options of Pioneer average runtimes for 10,000 generations was about 10 minutes on a SUN SPARC station2, and for the '*c*' option the runtime is about 10 minutes for 1,000 generations. The runtime for the '*c*' option was high, as in this case Pioneer needs to calculate the number of constraints satisfied for every population in every generation.

TABLE I
Statistics of FSM Samples

File	No. of states	No. of inputs	No. of outputs
bbara	10	8	6
bbsse	16	11	11
cse	16	11	11
beecount	7	6	7
sse	16	7	7
dk14	7	3	5
dk15	4	3	5
dk16	28	2	3
dk512	14	1	3
ex2	19	2	2
ex3	12	2	2
ex4	14	6	9
ex5	9	2	2
ex6	8	5	8
ex7	10	2	2
log	14	7	22
mark1	15	5	16
s8	5	4	1
train11	11	2	1
lion9	9	2	1

6 CONCLUSIONS

We have shown that evolution programming can be used effectively for the state assignment problem. The most distinct advantage of this method is that by carefully choosing a set of "genetic operators," the system can escape local optima, hence performing better than hill climbing based systems. A new tool was presented which provides three common cost functions. Although the cost functions are not as accurate as one would like, good results were obtained. Better results can be achieved with cost func-

TABLE II
Effectiveness of Pioneer on MCNC FSM Benchmarks

File	Mustang -p	Mustang -n	Pioneer -p	Pioneer -n	Pioneer -c
bbara	62	65	57	57	57
bbsse	151	132	115	113	113
cse	226	229	205	210	198
beecount	40	42	37	40	36
sse	151	132	103	123	107
dk14	111	102	85	102	77
dk15	64	60	63	61	60
dk16	285	271	244	224	239
dk512	67	71	52	54	54
ex2	181	158	152	122	118
ex3	76	78	73	53	63
ex4	76	85	74	68	57
ex5	64	68	55	51	51
ex6	115	122	104	104	90
ex7	73	73	65	59	67
log	117	109	112	95	110
mark1	92	88	70	65	61
s8	26	26	29	26	22
train11	52	42	31	26	39
lion9	23	16	15	18	22
TOTAL	**2052**	**1969**	**1741**	**1671**	**1641**

tions that better estimate the literal savings resulting after minimization using such tools as "MIS-II." However, one advantage of Pioneer is that the number of 1's in the code does not have to be estimated because it is known a priori as discussed in Section 4.

Four genetic operators were used in Pioneer: two mutation operators and two crossover operators. Fine tuning of these operators helped reduce the time required for convergence. Higher mutation rates were used compared to classical genetic algorithm to induce more randomness.

There are many different avenues for future research. One interesting avenue is incorporating more problem-specific knowledge into crossover. Since each generation of solutions is represented by a set of possible assignments, it is possible to vary the crossover between each of the assignments in the generation set. The current method of crossover in Pioneer can be improved upon by selecting from the

parents those states which are most strongly connected together. However, there is a speed penalty associated with that since the relationships between the states in every parent will have to be established based upon some literal savings cost estimator.

Although we experimented with varying the number of encoding bits (not shown), we did not achieve good results principally because the number of encoding bits, be it minimal or not, was fixed a priori. We are currently experimenting with building the number of encoding bits as a variable into Pioneer. This new variable will be tuned by the processes of crossover and mutation. It is not clear, at this stage, how this will affect the performance of Pioneer but theoretically it should help in some cases because increasing the number of encoding bits results in more freedom in assigning codes which can lead to satisfying a larger percentage of group constraints.

Other research directions include the use of a cost function derived from a multilevel model of the finite state machine to guide the evolution program. This cost function would be more accurate than the one used by MUSE because we have a priori knowledge of the code.

Acknowledgments

The authors would like to thank the reviewers of the paper for making a very thorough review. Their comments helped to improve the quality of this manuscript.

References

[1] D.B. Armstrong, "On the Efficient Assignment of Internal Codes to Sequential Machines," *IRE Trans. on Electronic Computers, EC(13),* Oct. 1962, pp. 661–672.

[2] R. Brayton, R. Rudell, A. Wang, and A. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis," Klumer Academic Press, Boston, MA, 1984.

[3] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "MIS: A Multilevel Logic Optimization System," *IEEE Trans. on Computer Aided Design, CAD(6),* November 1987.

[4] S. Coombs and L. Davis, "Genetic Algorithms and Communication Link Speed Design: Constraints and Operators," *Proc. of the Second International Conference on Genetic Algorithms,* Lawrence Erlbaum Associates, Hillsdale, NJ, 1987, pp. 257–260.

[5] L. Davis, *Genetic Algorithms and Simulated Annealing,* Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1987.

[6] L. Davis, "Adapting Operator Probabilities in Genetic Algorithms," *Proceedings of the Third International Conference on Genetic Algorithms,* Morgan Kaufman Publishers, Los Altos, CA, 1989, pp. 61–69.

[7] G. DeMicheli, R.K. Brayton, and A. Sangiovanni-Vincentelli, "Optimal State Assignment for Finite State Machines," *IEEE Trans. Computer-Aided Design,* July 1985, pp. 269–285.

[8] D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning,* Addison Wesley, Reading, MA, 1989.

[9] J. Hartmanis, "On the State Assignment Problem for Sequential Machines I," *IRE Trans. on Elect. Comp.,* June 1961, pp. 157–165.

[10] J. Hartmanis and R.E. Stearns, *Algebraic Structure Theory of Sequential Machines,* Prentice Hall, 1966.

[11] G.D. Hachtel, M. Lightner, R. Jocoby, C. Morrison, P. Moceyunas, and D. Bostrick. "Bold: The boulder optimal logic design system," In *Hawaii International symposium on Systems Sciences,* 1988.

[12] J. Holland, *Adaptation in Natural and Artificial Systems,* University of Michigan Press, Ann Arbor, 1975.

[13] R. Karp, "Some Techniques for State Assignment for Synchronous Sequential Machines," *IEEE Trans. on Electronic Computers,* June 1964, pp. 507–518.

[14] Z. Kohavi, "Secondary State Assignment for Sequential Machines," *IEEE Trans. on Electronic Computers,* June 1964, pp. 193–203.

[15] B. Lin and A.R. Newton, "Synthesis of Multiple Level Logic From Symbolic High-Level Description Languages," *Proc. IFIP Int. Conf. on VLSI,* August 1991, pp. 187–196.

[16] R.Z. Makki, J. Muha, S. Boughazale, and T. Kaylani, "SSC: A Tool for the Synthesis of Testable Sequential Machines," *Proc. of COMPCON,* Spring 1990, pp. 455–461.

[17] E.J. McCluskey and T.A. Dolatta," Coding of Internal Codes to Finite State Machines," *IRE Trans. on Electronic Computers, EC(13),* Oct. 1962, pp. 549–562.

[18] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs,* Springer Verlag, New York, 1992.

[19] Z. Michalewicz and C. Janikow, "GENECOP: A Genetic Algorithm for Numerical Optimization Problems with Linear Constraints," to appear in *Communications of the ACM,* 1992.

[20] D.J. Montana and L. Davis, "Training Feedforward Neural Networks Using Genetic Algorithms," *Proc. of the 1989 International Joint Conference on Artificial Intelligence,* Morgan Kaufmann Publishers, Los Altos, CA, 1989.

[21] S. F. Smith, "A Learning System Based on Genetic Algorithms," Ph.D. dissertation, University of Pittsburgh, 1980.

[22] S. Devadas, Hi-K T. Ma, A.R. Newton, and A. Sangiovanni-Vincentelli, "MUSTANG: State Assignment of Finite State Machines for Optimal Multi Level Logic Implementations," *Proc. Int. Conf. on Computer-Aided Design,* Nov. 1987, pp. 16–19.

[23] S. Devadas, Hi-K T. Ma, A.R. Newton, and A. Sangiovanni-Vincentelli, "Synthesis and Optimization Procedures for Fully and Easily Testable Sequential Machines," *Proc. Int. Test Conf.,* Sept. 1988, pp. 621–630.

[24] S. Su and Rafic. Z. Makki, "Analysis and Characterization of State Assignment Techniques for Sequential Machines," *MCNC Technical Report, TR90-37.*

[25] T. Villa and A. Sangiovanni-Vincentelli, "NOVA: State Assignment of Finite State Machines for Optimal Two-Level Logic Implementations," *Proc. 26th Design Automation Conference,* June 1989, pp. 327–332.

[26] G.A. Vignaux and Z. Michalewicz, "A Genetic Algorithm for the Linear Transportation Problem," *IEEE Transactions on Systems, Man and Cybernetics,* Vol. 21, No. 2, pp. 445–452, 1991.

[27] X. Du, G. Hachtel, and P. Moceyunas, "MUSE: A MUltilevel Symbolic Encoding Algorithm for State Assignment," *Hawaii Int. Conf. on Sys. Science,* Jan. 1990.

Biographies

SUBBU MUDDAPPA received his B.S. degree in Electrical Engineering from Bangalore University in 1990 and his M.S. in Electrical Engineering from the University of North Carolina at Charlotte in 1993. He currently holds the position of test engineer with Cirrus Logic. His technical interests include logic synthesis, test program development, and ATE.

RAFIC MAKKI received his B.E. and M.S. degrees in Electrical Engineering from Youngstown State University in 1979 and 1980 respectively. He received his Ph.D. degree in Electrical Engineering from Tennessee Technological University in 1983. He currently holds the position of associate professor and director of graduate programs in Electrical Engineering at the University of North Carolina at Charlotte. His technical interests include logic synthesis, design for testability, power supply current testing, and fault modeling.

ZBIGNIEW MICHALEWICZ received his M.S. degree from the Technical University of Warsaw in 1974, and Ph.D. in Computer Science from the Institute of Computer Science, Polish Academy of Sciences, in 1981. Currently he is Associate Professor at the Department of Computer Science, University of North Carolina at Charlotte. His major research interests include evolutionary computation methods, optimization, and database systems.

SRIDHAR ISUKAPALLI received his B.E. in Computer Science and Engineering from Mysore University in 1988 and his M.S. in Computer Science from the University of North Carolina at Charlotte in 1991. He is currently a design engineer with Alliance Semiconductor. His technical interests include VLSI design, computer architecture, and microprocessor systems design.