Evolutionary Synthesis of Synchronous Finite State Machines

Nadia Nedjah and Luiza de Macedo Mourelle

Department of Systems Engineering and Computation, Faculty of Engineering,
State University of Rio de Janeiro,
Rio de Janeiro, Brazil
{nadia, ldmm}@eng.uerj.br
http://www.eng.uerj.br/~ldmm

Abstract - **Synchronous finite state machines are very important for digital sequential designs. They allow the synchronisation of the hardware system components so that these may cooperate adequately in the fulfilment of the main objective of the design. In this paper, assuming that the state assignment problem has been solved and so a specific state coding is provided, we propose to use the evolutionary methodology to yield optimal evolvable hardware that implement the state machine control component. The evolved hardware requires a minimal hardware area and introduces a minimal propagation delay of the machine output signals.**

I. INTRODUCTION

Sequential digital systems or simply finite state machines have two main characteristics: *(i)* there is at least one feedback path from the system output signal to the system input signals; and *(ii)* there is a memory capability that allows the system to determine current and future output signal values based on the previous input and output signal values [12]. Traditionally, the design process of a state machine passes through five main steps:

1. Specification of the sequential system, which should determine the next states and outputs of every present state of the machine. This is done using state tables and state diagrams;
2. State reduction, which should reduce the number of present states using equivalence and output class grouping;
3. State assignment, which should assign a distinct combination to every present state. This may be done using Armstrong-Humphrey heuristics [12];
4. Minimisation of the control combinational logic using K-maps and transition maps;
5. Implementation of the state machine, using gates and flip-flops.

In this paper, we concentrate on the fourth step of the design process, i.e. the minimisation of the control combinational logic using genetic programming. For this purpose, we assume that the best state assignment has been found and is provided to the evolutionary process.

Designing a hardware that fulfils a certain function consists of deriving from specific input/output behaviours, an architecture that is *operational* (i.e. produces all the expected outputs from the given inputs) within a specified set of constraints. Besides the input/output behaviour of the hardware, conventional designs are essentially based on knowledge and creativity, which are two human characteristics and too hard to be automated. Evolutionary hardware is a design that is generated using simulated evolution as an alternative to conventional-based electronic circuit design. *Genetic evolution* [9] is a process that evolves a set of genotypes, i.e. *population*, producing a new population at each iteration process. Here, individuals are hardware designs. The more the design obeys the constraints, the more it is used in the reproduction process. The design constraints can be expressed in terms of hardware area and/or response time requirements. The freshly produced population is yield using some *genetic operators* such as *crossover* and *mutation* that attempt to simulate the natural breeding process in the hope of generating new design that are *fitter* i.e. respect more the design constraints. Genetic evolution is usually implemented using *genetic algorithms*.

Genetic programming [5], [9] is way of producing a program using genetic evolution. So the individuals are programs. The main goal of genetic programming is to provide a domain-independent problem-solving method that automatically yields computer programs from expected input/output behaviours. Exploiting genetic programming, we automatically create novel control logic circuits that exhibit creativity and inventiveness. Here, the individuals are *register-transfer level* specifications of the hardware. All existing synthesis tools synthesize this kind of specifications very quickly, and so an equivalent re-configurable hardware can be obtained in a matter of seconds.

The remainder of this paper is organised into five sections. In Section 2, we introduce the problems that face the designer of finite state machine, which are mainly the state assignment problem and the control logic. We show that a better assignment improves considerably the cost of the control logic. In Section 3, we give an overview on evolutionary computations and genetic algorithms and their application to engineer evolvable hardware and we design a genetic programming-based synthesiser for evolving minimal control logic circuit provided the state assignment for the specification of the state machine in question. We describe the genetic operators used as well as the fitness function, which determines whether a state assignment is better that another and how much. In Section 4, we present results evolved through our genetic algorithm for some well-known benchmarks and compare the obtained results with those obtained using the traditional method to design state machine, i.e. using Karnaugh maps and flip-flop transition maps.

II. STATE ASSIGNMENT AND CONTROL LOGIC

Once the specification and the state reduction step have been completed, the next step consists then of assigning a code to each state present in the machine. It is clear that if the machine has N distinct states then one needs N distinct combinations of 0s and 1s. So one needs K flip-flops to store the machine current state, wherein K is the smallest positive integer such that $2^K \geq N$. The state assignment problem under consideration consists of finding the *best* assignment of the flip-flop combinations to the machine states. Since a machine state is nothing but a counting device, combinational control logic is necessary to activate the flip-flops in the desired sequence. This is shown in Fig. 1, wherein the feedback signals constitute the machine state, the control logic is a combinational circuit that computes the state machine output signals (also called *primary output signals*) from the state signals (also called *current state*) and the input signals (also called *primary input signals*). It also produces the signals of new machine state (also called *next state*).

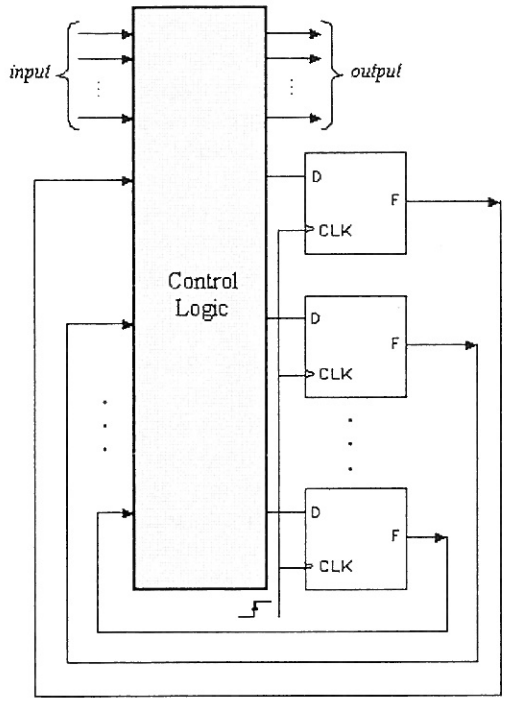

Fig. 1. The subcomponents of a finite synchronous state machine

The control logic component in a state machine is responsible of generating the primary output signals as well as the signal that form the next state. It does so using the primary input signals and the signals that constitute the current state (see Fig. 1). Traditionally, the combinational circuit of the control logic is obtained using the transition maps of the flip-flops [12].

Given a state transition function, it is expected that the complexity (area and time) and so the cost of the control logic will vary for different assignments of flip-flop combinations to allowed states. Consequently, the designer should seek the assignment that minimises the complexity and so the cost of the combinational logic required to control the state transitions. For instance, consider the state machine of one input signal (I), one output signal (O) and 4 states whose state transition function is given in tabular form in TABLE I and assume that we use D-flip-flops to store the machine current state. Then the state assignment $A_0 = \{s_0 \equiv 00, s_1 \equiv 11, s_2 \equiv 01, s_3 \equiv 10\}$ requires a control logic that consists of 3 NOT gates, 5 AND gates and 3 OR gates while the assignments $A_1 = \{s_0 \equiv 00, s_1 \equiv 10, s_2 \equiv 01, s_3 \equiv 11\}$ requires a control logic that consists of only 2 NOT gates, 5 AND gates and 2 OR gates. The schematics of the state machines that encode the state according to state assignments A_0 and A_1 are given in Fig. 2 and Fig. 3 respectively.

TABLE I. STATE TRANSITION FUNCTION

Present State	Next Sate		Output (O)	
	$I=0$	$I=0$	$I=1$	$I=1$
q_0	q_0	q_0	0	0
q_1	q_2	q_2	0	1
q_2	q_0	q_0	1	0
q_3	q_2	q_2	1	1

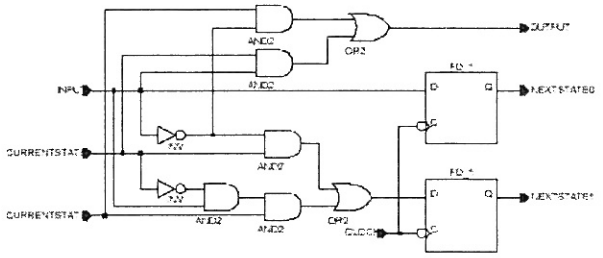

Fig. 2. The machine state schematics for state assignment A_0

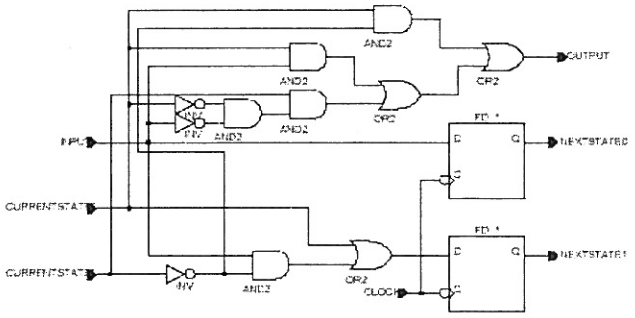

Fig. 3. The machine state schematics for state assignment A_1

In this paper we focus on evolving minimal control logic for state machines. We assume that the best state assignment has been found. It is clear that finding the best assignment is an *NP*-complete problem. It can be solved using genetic algorithms [2], [3], [8], [11].

III. EVOLVABLE HARDWARE FOR THE CONTROL LOGIC

Genetic programming [8], [9] is an extension of genetic algorithms. The chromosomes are computer programs and the genes are instructions. In general, genetic programming offers a mechanism to get a computer to provide a solution of problem without being told exactly how to do it. In short, it allows one to automatically create a program. It does so based on a high level statement of the constraints the yielded program should obey to. The input/output

behaviour of the expected program is generally considered as an omnipresent constraint. Furthermore, the generated program should use a minimal number of instructions and have an optimal execution time.

Starting form random set of computer programs, which is generally called *initial population*, genetic programming breeds a population of programs through a series of steps, called *generations*, using the Darwinian principle of natural *selection*, re-combination also called *crossover*, and *mutation*. Individuals are selected based on how much they adhere to the specified constraints. Each program is assigned a value, generally called its *fitness*, which mirrors how *good* it is in solving the program. Genetic programming [9] proceeds by first, randomly creating an initial population of computer programs; then, iteratively performing a generation, which consists of going through two main steps, as far as the constraints are not met. The first step in a generation assigns for each computer program in the current population a fitness value that measures its adherence to the constraints while the second step creates a new population by applying the three genetic operators, which are *reproduction*, *crossover* and *mutation* to some selected individuals. *Selection* is done with on the basis of the individual fitness. The fitter the program is, the more probable it is selected to contribute to the formation of the new generation. *Reproduction* simply copies the selected individual from the current population to the new one. *Crossover* recombines two chosen computer programs to create two new programs using single-point crossover or two-points crossover [5]. Mutation yields a new individual by changing some randomly chosen instruction in the selected computer program. The number of genes to be mutated is called *mutation degree* and how many individuals should suffer mutation is called *mutation rate*.

There three main aspects in implementation of genetic programming [5], [8], [9]: *(i)* program encoding; *(ii)* crossover and mutation of programs; *(iii)* program fitness. In this section, we explain how we treat these three aspects in our implementation.

A. Circuit Encoding

Encoding of individuals is one of the implementation decisions one has to take in order to use evolutionary computation. It depends highly on the nature of the problem to be solved. There are several representations that have been used with success [10]: *binary encoding* which is the most common mainly because it was used in the first works on genetic algorithms, represents an individual as a string of bits; *permutation encoding* mainly used in ordering problem, encodes an individual as a sequence of integer; *value encoding* represents an individual as a sequence of values that are some evaluation of some aspect of the problem; and *tree encoding* represents an individual as tree. Generally, the tree coincides with the *concrete tree* as opposed to *abstract tree* [1] of the computer program, considering the grammar of the programming language used.

Here a design is specified using register transfer level equations. Each instruction in the specification is an output signal assignment. A signal is assigned the result of an expression wherein the operators are those that represent basic gates in CMOS technology of VLSI circuit implementation and the operands are the input signals of the design. The allowed operators are shown in TABLE II. Note that all gates introduce a minimal propagation delay as the number of input signal is minimal, which is 2.

TABLE II. GATE NAME, SYMBOL, GATE EQUIVALENT, DELAY

Name	Symbol	Gate Code	Gate Equiv.	Delay
NOT		0	1	0.0625
AND		1	2	0.209
OR		2	2	0.216
XOR		3	3	0.212
NAND		4	1	0.13
NOR		5	1	0.156
XNOR		6	3	0.211
MUX		7	3	0.212

We encode circuit schematics using a matrix of cells that may be interconnected. A cell may or may not be involved in the circuit schematics. A cell consists of two inputs or three in the case of a MUX, a logical gate and a single output. A cell may draw its input signals from the output signals of gates of previous rows. The gates include in the first row draw their inputs from the circuit global input signal or their complements. The circuit global output signals are the output signals of the gates in the last raw of the matrix. A chromosome with respect to this encoding is given in TABLE III. It represents the circuit of Fig. 4. Note that the input signals are numbered 0 to 3, their negated signals are numbered 4 to 7 and the output signals are numbered 16 to 19. If the circuit has n outputs with $n < 4$, then the signals numbered 16 to n are the actual output signals of the circuit.

TABLE III. CHROMOSOME FOR THE CIRCUIT OF FIG. 4

⟨1,0, 2, 8⟩	⟨5, 10, 9,12⟩	⟨7,13, 14, 11, 16⟩
⟨2, 4, 3, 9⟩	⟨1, 8,10, 13⟩	⟨3, 11, 12, 17⟩
⟨3, 1, 6, 10⟩	⟨4, 9, 8, 14⟩	⟨7, 15, 14, 15, 18⟩
⟨7, 5, 7, 7, 11⟩	⟨4, 10, 11, 15⟩	⟨1, 11, 15, 19⟩

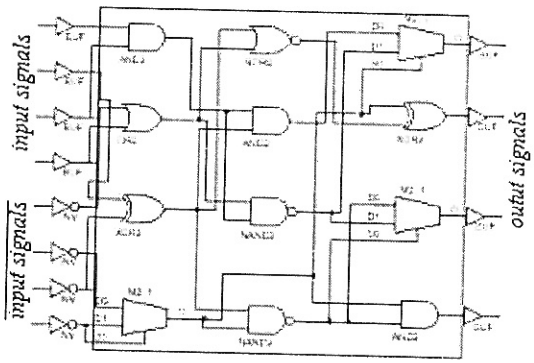

Fig. 4. Encoded circuit

B. Circuit Reproduction

Crossover recombines two randomly selected individuals into two fresh offspring. It may be *single-point* or *double-point* or *uniform* crossover [8]. Crossover of circuit specification is implemented using a variable four-point crossover as described in Fig. 5.

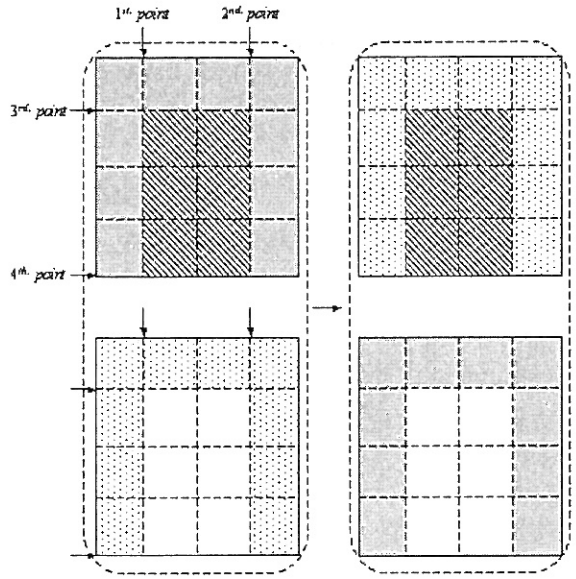

Fig. 5. Four-point crossover of circuit schematics

One of the important and complicated operators for genetic programming is the *mutation*. It consists of changing a gene of a selected individual. The number of individuals that should suffer mutation is defined by the *mutation rate* while how many genes should be altered within a chosen individual is given by the *mutation degree*. Here, a gene is the expression tree on the left hand side of a signal assignment symbol. Altering an expression can be done in two different ways depending the node that was randomised and so must be mutated. A node represents either an operand or operator. In the former case, the operand, which is a bit in the input signal, is substituted with either another input signal or *simple* expression that includes a single operator as depicted in Fig. 6 – top part. The decision is random. In the case of mutating an operand node to an operator node, we proceed as Fig. 6 – bottom part. The randomised operator node may be mutated to an operator node or to an operator of smaller (AND to NOT), the same (AND to XOR) or bigger arity (AND to MUX). In the last case, a new operand is randomised to fill in the new operand.

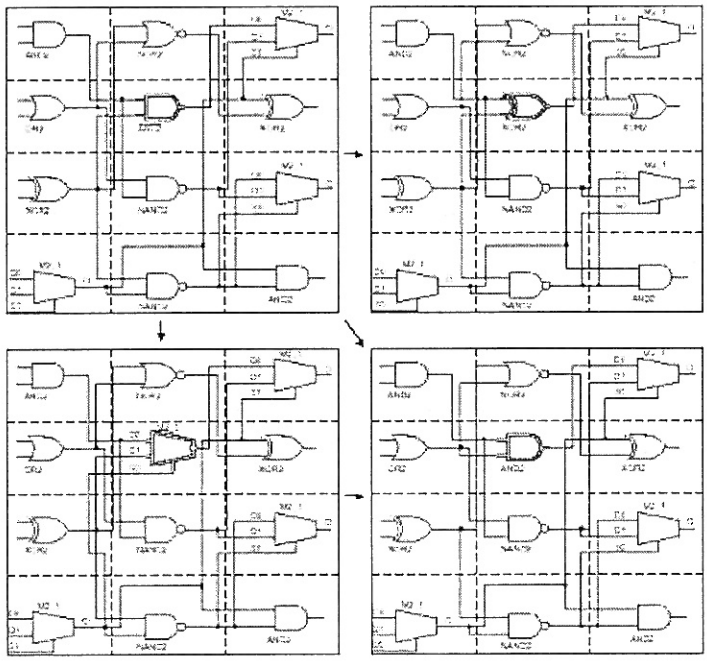

Fig. 6. Operand node mutation for circuit specification

C. Circuit Evaluation

Another important aspect of genetic programming is to provide a way to evaluate the adherence of evolved computer programs to the imposed constraints. In our case, these constraints are of three kinds. First of all, the evolved specification must obey the input/output behaviour, which is given in a tabular form of expected results given the inputs. This is the truth table of the expected circuit. Second, the circuit must have a reduced size. This constraint allows us to yield compact digital circuits. Thirdly, the circuit must also reduce the signal propagation delay. This allows us to reduce the response time and so discover efficient circuits. In order to take into account both area and response time, we evaluate circuits using the weighted sum approach.

We estimate the necessary area for a given circuit using the concept of *gate equivalent*. This is the basic unit of measure for digital circuit complexity [6]. It is based upon the number of logic gates that should be interconnected to perform the same input/output behavior. This measure is more accurate that the simple number of gates [6], [12].

When the input to an electronic gate changes, there is a finite time delay before the change in input is seen at the output terminal. This is called the propagation delay of the gate and it differs from one gate to another. Of primary concern is the path from input to output with the highest total propagation delay. We estimate the performance of a given circuit using the worst-case delay path. The number of gate equivalent and an average propagation delay for each kind of gate are given in TABLE I. The data were taken form [6].

Let C be a digital circuit that uses a subset (or the complete set) of the gates given in TABLE I. Let *Gates*(C) be a function that returns the set of all gates of circuit C and *Levels*(C) be a function that returns the set of all the gates of C grouped by level. Notice that the number of levels of a circuit coincides with the cardinality of the set expected from function *Levels*. On the other hand, let *Val*(X) be the Boolean value that the considered circuit C propagates for the input Boolean vector X assuming that the size of X coincides with the number of input signal required for circuit C. The fitness function, which allows us to determine how much an evolved circuit adheres to the specified constraints, is given as follows, wherein X represents the input values of the input signals while Y represents the expected output values of the output signals of circuit C, n denotes the number of output signals that circuit C has, function *Delay* returns the propagation delay of a given gate as shown in TABLE I and Ω_1 and Ω_2 are the weighting coefficients [7] that allow us to consider both area and response time to evaluate the performance of an evolved circuit, with $\Omega_1 + \Omega_2 = 1$. For implementation issue, we minimized the fitness function below for different values of Ω_1 and Ω_2.

$$Fitness(C) = \sum_{j=1}^{n}\left(\sum_{i|Val(X_i)\neq Y_{i,j}} Penalty\right) + \Omega_1 \times \sum_{g\in Gates(C)} GateEquiv(g) + \Omega_2 \times \sum_{L\in Levels(C)} \max_{g\in L} Delay(g)$$

IV. COMPARATIVE RESULTS

In this section, we compare the evolved circuits to those obtained using the traditional methods, i.e. transition and Karnaugh maps. This is done for three different state machines that are generally used as benchmarks. These state machines are commonly called *shiftreg*, *lion*9 and *train*11. The detailed descriptions of these state machines can be found in [4]. The state assignments used are the best ones found so far. They also are the result of an evolutionary computation [11]. Theses state assignment are given in TABLE IV, wherein *#S*, *#T*, *#In* and *#Out* stand for the number of states of the machine, the number of state transitions, the number of primary input signals and the number of primary output signals respectively.

TABLE IV. THE STATE MACHINES USED AS BENCHMARKS

FSM	#S	#T	#In	#Out	State assignment
Shiftreg	8	16	1	1	[4, 0, 3, 7, 5, 1, 2, 6]
*Lion*9	9	25	2	1	[10, 8, 12, 9, 13, 15, 7, 3, 11]
*Train*11	11	25	2	1	[2, 6, 1, 4, 0, 14, 10, 9, 8, 11, 3]

For each of the state machines described in TABLE IV we evolved a minimal circuit that implements the required behaviour and compared it to the one engineered using the traditional method. TABLE V shows the details of this comparison. The schematics of the evolved circuit of the considered state machines are given in Fig. 7, Fig. 8 and Fig. 9 respectively. It is clear that the evolved circuits are much better that those yield by the traditional methods in both terms hardware area and signal propagation delay.

TABLE V. COMPARISON OF THE TRADITIONAL METHOD VERSUS GENETIC PROGRAMMING

State machine	#gate-Equiv		Response time	
	Traditional	*GP*	*Traditional*	*GP*
Shiftreg	30	12	0.85	0.423
*Lion*9	102	33	2.513	0.9185
*Train*11	153	39	2.945	0.8665

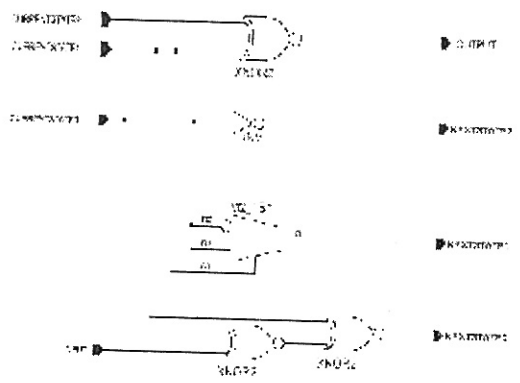

Fig. 5. First evolved control logic for state machine *shiftreg*

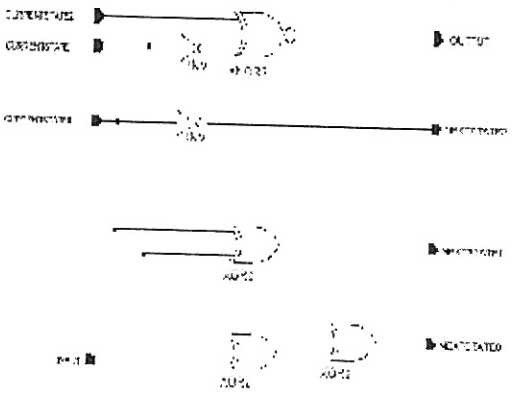

Fig. 6. Second evolved control logic for state machine *shiftreg*

V. CONCLUSION

In this paper, we exploited evolutionary computation to synthesise the control logic used in asynchronous finite state machines. We compared the circuits evolved by our genetic programming-based synthesiser with that that would use the traditional method, i.e. using Karnaugh maps and transition maps. The state machine used as benchmarks are well known and of different sizes. Our genetic algorithm always obtains better control logic both in terms of hardware area required to implement the circuit and response time (see TABLE V).

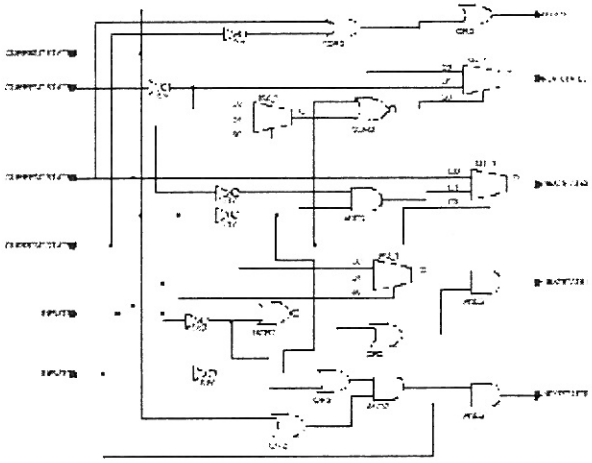

Fig. 6. The evolved control logic for state machine *lion9*

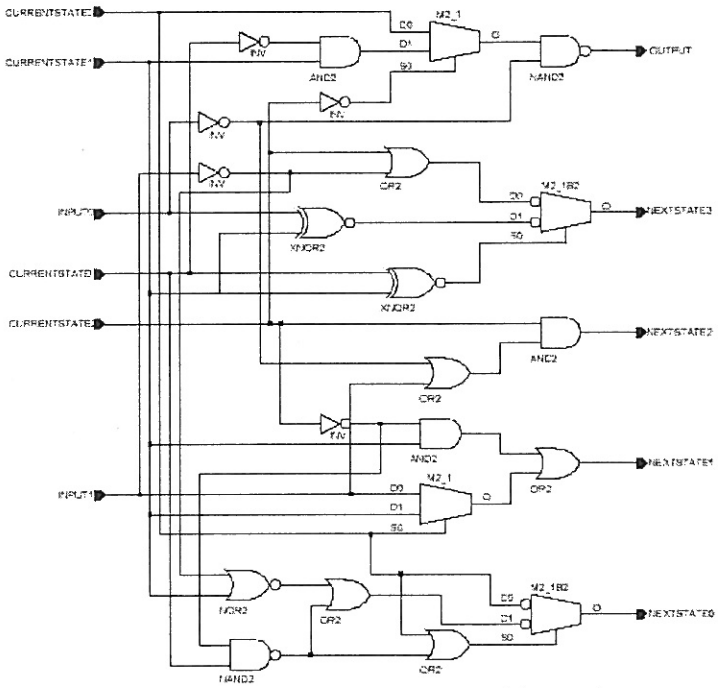

Fig. 9. The evolved control logic for state machine *train11*

VI. REFERENCES

[1] Aho, A.V. , S. Ravi and J.D. Ullman, *Compilers: principles, techniques and tools*, Addison-Wesley, 1986.

[2] Amaral, J.N., Tumer, K. and Gosh, J., *Designing genetic algorithms for the State Assignment problem*, IEEE Transactions on Systems Man and Cybernetics, vol., no. 1999.

[3] Armstrong, D.B., *A programmed algorithm for assigning internal codes to sequential machines*, IRE Transactions on Electronic Computers, EC 11, no. 4, pp. 466-472, August 1962.

[4] Collaborative Benchmarking Laboratory, North Carolina State University,`http://www.cbl.ncsu.edu/pub/Benchmark_dirs/LGSynth89/fsmexamples/`, November 2003.

[5] DeJong, K. and Spears, W.M., *Using genetic algorithms to solve NP-complete problems*, Proc. of the Third International Conference on Genetic Algorithms, pp. 124-132, Morgan Kaufmann, 1989.

[6] Ercegovac, M. D., Lang, T. and Moreno, J.H., *Introduction to digital systems*, John Wiley, 1999.

[7] Fonseca, C.M. and Fleming, P.J., *An overview of evolutionary algorithms in multi-objective optimization*, Evolutionary Computation, 3(1):1-16.

[8] Haupt, R.L. and Haupt, S.E., *Practical genetic algorithms*, John Wiley and Sons, 1998.

[9] Koza, J.R., *Genetic Programming*. MIT Press, 1992.

[10] Michalewics, Z., *Genetic algorithms + data structures = evolution program*, Springer-Verlag, USA, third edition, 1996.

[11] Nedjah, N. and Mourelle, L.M, *Evolutionary state assignment for synchronous finite state machine*, Proc. of International Conference on Computational Science, LNCS, Springer-Verlag, 2004.

[12] Rhyne, V.T., *Fundmentals of digital systems design*, Computer Applications in Electrical Engineering Series, Prentice-Hall, 1973.