Evolutionary State Assignment for 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. Among other important aspects, finite state machines represent a powerful way for synchronising hardware components so that these components may cooperate in the fulfilment of the main objective of the hardware design. In this paper, we propose to optimally solve the state assignment *NP*-complete problem, which is inherent to designing any synchronous finite state machines using evolutionary computations. This is motivated by to reasons: first, finite state machines are very important in digital design and second, with an optimal state assignment one can physically implement the state machine in question using a minimal hardware area and reduce the propagation delay of the machine output signals.

1 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 [1].

Traditionally, the design process of a state machine passes through five main steps: *(i)* the 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; *(ii)* the state reduction, which should reduce the number of present states using equivalence and output class grouping; *(iii)* the state assignment, which should assign a distinct combination to every present state. This may be done using Armstrong-Humphrey heuristics [1]; *(iv)* the minimisation of the control combinational logic using K-maps and transition maps; *(v)* and finally, the implementation of the state machine, using gates and flip-flops.

In this paper, we concentrate on the third step of the design process, i.e. the state assignment problem. We present a genetic algorithm designed for finding a state assignment of a given synchronous finite state machine, which attempts to minimise the cost related to the state transitions.

The remainder of this paper is organised into five sections. In Section 2, we explain thoroughly the state assignment problem and show that a better assignment improves

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3038, pp. 1289–1296, 2004.

considerably the cost related to state transitions. In Section 3, we give an overview on evolutionary computations and genetic algorithms and their application to solve *NP*-problems. In Section 4, we design a genetic algorithm for evolving best state assignment for a given state machine specification. 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 5, we present results evolved through our genetic algorithm for some well-known benchmarks. Then we compare the obtained results with those obtained by another genetic algorithm described in [2] as well as with NOVA, which is uses well established but non-evolutionary method [3].

2 State Assignment Problem

Once the specification and the state reduction step has been completed, the next step is then to assign a code to each state present in the machine. It is clear that if the machine has N present states then, one needs N distinct combinations of 0s and 1s. So one needs K flip-flops to store the machine present 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. Given a state transition function, it is expected that the complexity (area and time) as well as 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 the cost of the combinational logic required to control the state transitions. For instance, consider the state machine of one input signal and 4 states whose state transition function is given in tabular form in Table 1 and we are using JK-flip-flops to store the machine current state. Then the state assignment $\{s_0 \equiv 00, s_1 \equiv 10, s_2 \equiv 01, s_3 \equiv 11\}$ requires a control logic that consists of 4 NOT gates, 3 AND gates and 1 OR gate while the assignments $\{s_0 \equiv 00, s_1 \equiv 11, s_2 \equiv 01, s_3 \equiv 10\}$, $\{s_0 \equiv 10, s_1 \equiv 01, s_2 \equiv 11, s_3 \equiv 00\}$ and $\{s_0 \equiv 01, s_1 \equiv 10, s_2 \equiv 00, s_3 \equiv 11\}$ require a control logic that consists of only 2 NOT gates and 1 OR gate.

Table 1. State transition function

Present State	Next Sate	
	$I = 0$	$I = 1$
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_0	q_3
q_3	q_2	q_1

3 Evolutionary Computations

Evolutionary algorithms are computer-based solving systems, which use evolutionary computational models as key element in their design and implementation. A variety of evolutionary algorithms have been proposed. The most popular ones are *genetic algorithms* [4]. They have a conceptual base of simulating the evolution of individual structures via the Darwinian natural selection process. The process depends on the adherence of the individual structures as defined by its environment to the problem pre-determined constraints. Genetic algorithms are well suited to provide an efficient solution of *NP*-hard problems [5].

Genetic algorithms maintain a *population* of *individuals* that evolve according to *selection* rules and other *genetic operators*, such as *mutation* and *recombination*. Each individual receives a measure of *fitness*. *Selection* focuses on individuals, which shows high fitness. *Mutation* and *crossover* provide general heuristics that simulate the *recombination* process. Those operators attempt to perturb the characteristics of the parent individuals as to generate *distinct* offspring individuals.

Genetic algorithms are implemented through the following generic algorithm described by Algorithm 1, wherein parameters *ps*, *f* and *gn* are the population size, the fitness of the expected individual and the number of generation allowed respectively.

Algorithm 1. `GA(ps, f, gn):individual;`

```
1: generation  := 0;
2: population  := initialPopulation();
3: fitness := evaluate(population);
4: do parents   := select(population);
5:    population := reproduce(parents);
6:    fitness    := evaluate(population);
7:    generation := generation + 1;
8: while(fitness[i] < f, ∀ i ∈ population) & (generation < gn);
9: return fittestIndividual(population);
End
```

In Algorithm 1, function *intialPopulation* returns a valid random set of individuals that compose the population of the first generation, function *evaluate* returns the fitness of a given population. Function *select* chooses according to some criterion that privileges fitter individuals, the individuals that will be used to generate the population of the next generation and function *reproduction* implements the crossover and mutation process to yield the new population.

4 Application to the State Assignment Problem

The identification of a good state assignment has been thoroughly studied over the years. In particular, Armstrong [6] and Humphrey [7] have pointed out that an assignment is good if it respects two rules, which consist of the following: *(i)* two or more states that have the same next state should be given adjacent assignments; *(ii)* two or more states that are the next states of the same state should be given adjacent assignment. State adjacency means that the states appear next to each other in the mapped representation. In other terms, the combination assigned to the states should

differ in only one position; and *(iii)* the first rule should be given more important the second. For instance, state codes 0101 and 1101 are adjacent while state codes 1100 and 1111 are not adjacent.

Now we concentrate on the assignment encoding, genetic operators as well as the fitness function, which given two different assignment allows one to decide which is fitter.

4.1 Assignment Encoding

Encoding of individuals is one of the implementation decisions one has to make in order to use genetic algorithms. It very depends on the nature of the problem to be solved. There are several representations that have been used with success [4].

In our implementation, an individual represents a state assignment. We use the *integer encoding*. Each chromosome consists of an array of N entries, wherein entry i is the code assigned to $i^{th.}$ machine state. For instance, chromosome in Fig. 1 represents a possible assignment for a machine with 6 states:

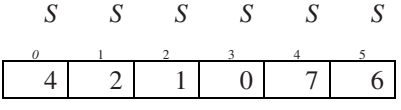

Fig. 1. State assignment encoding

4.2 The Individual Reproduction

Besides the parameters, which represent the population size, the fitness of the expected result and the maximum number of generation allowed, the genetic algorithm has several other parameters, which can be adjust by the user so that the result is up to his or her expectation. The selection is performed using some *selection probabilities* and the reproduction, as it is subdivided into *crossover* and *mutation* processes, depends on the kind of crossover and the mutation rate and degree to be used.

Given the parents populations, the reproduction proceeds using replacement as a reproduction scheme, i.e. offspring replace their parents in the next generation. Obtaining offspring that share some traits with their corresponding parents is performed by the *crossover* function. There are several *types* of crossover schemes [4]. The newly obtained population can then suffer some mutation, i.e. some of the genes of some of the individuals. The crossover type, the number of individuals that should be mutated and how far these individuals should be altered are set up during the initialisation process of the genetic algorithm.

There are many ways how to perform crossover and these may depend on the individual encoding used [4]. We present crossover techniques used with binary, permutation and value representations. *Single-point crossover* consists of choosing randomly one *crossover point*, then, the part of the bit or integer sequence from beginning of offspring till the crossover point is copied from one parent, the rest is copied from the second parent. *Double-points crossover* consists of selecting

randomly two *crossover points*, the part of the bit or integer sequence from beginning of offspring to the first crossover point is copied from one parent, the part from the first to the second crossover point is copied from the second parent and the rest is copied from the first parent. *Uniform crossover* copies integers randomly from the first or from the second parent. Finally, *arithmetic crossover* consists of applying some arithmetic operation to yield a new offspring.

The single point and two points crossover use randomly selected crossover points to allow variation in the generated offspring and to avoid premature convergence on a local optimum [4]. In our implementation, we tested single-point and double-point crossover techniques.

Mutation consists of changing some genes of some individuals of the current population. The number of individuals that should be mutated is given by the parameter *mutation rate* while the parameter *mutation degree* states how many genes of a selected individual should be altered. The mutation parameters have to be chosen carefully as if mutation occurs very often then the genetic algorithm would in fact change to *random search* [4]. Fig. 2 illustrates the genetic operators.

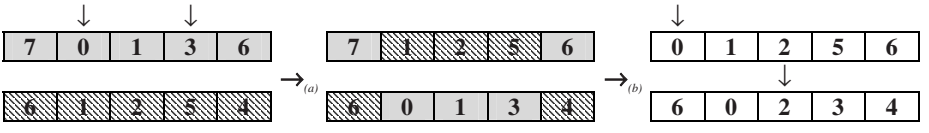

Fig. 2. State assignment genetic operators: *(a)* double-point crossover and *(b)* mutation

4.3 The Fitness Evaluation

This step of the genetic algorithm allows us to classify the individuals of a population so that fitter individuals are selected more often to contribute in the constitution of a new population. The fitness evaluation of state assignments is performed with respect to two rules of Armstrong [6] and Humphrey [7]: *(i)* how much a given state assignment adheres to the first rule, i.e. how many states in the assignment, which have the same next state, have no adjacent state codes; *(ii)* how much a given state in the assignment adheres to the second rule, i.e. how many states in the assignment, which are the next states of the same state, have no adjacent state codes.

In order to efficiently compute the fitness of a given state assignment, we use an *N*×*N adjacency matrix*, wherein *N* is the number of the machine states. The triangular bottom part of the matrix holds the expected adjacency of the states with respect o the first rule while the triangular top part of it holds the expected adjacency of the states with respect to the second rule. The matrix entries are calculated as in Equation (1), wherein *AM* stands for the adjacency matrix, functions *next*(*s*) and *prev*(*s*) yield the set of states that are next and previous to state *s* respectively. For instance, for the state machine in Table 1, we get the 4×4 adjacency matrix in Fig 3.

$$AM_{i,j} = \begin{cases} \#\left(next(q_i) \cap next(q_j)\right) & i > j \\ \#\left(prev(q_i) \cap prev(q_j)\right) & i < j \\ 0 & i = j \end{cases} \quad (1)$$

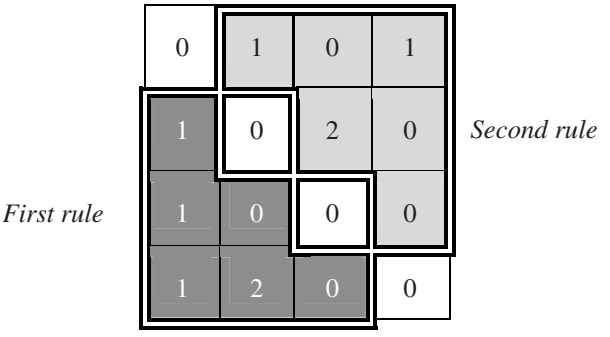

Fig. 3. Adjacency matrix for the machine state in Table 1

Using the adjacency matrix *AM*, the fitness function applies a penalty of 2, respectively 1, every time the first rule, respectively the second rule, is broke. Equation (2) states the details of the *fitness* function applied to a state assignment *SA*, wherein function *nadjacent* (*q*, *p*) returns 0 if the codes representing states *q* and *p* are adjacent and 1 otherwise.

$$fitness(SA) = \sum_{i=0}^{N-1} \sum_{j=i+1}^{N-1} \left(AM_{i,j} + 2 \times AM_{j,i}\right) \times nadjacent\left(SA_i, SA_j\right) \quad (2)$$

For instance, considering the state machine whose state transition function is described in Table 1, the state assignment $\{s_0 \equiv 00, s_1 \equiv 10, s_2 \equiv 01, s_3 \equiv 11\}$ has a fitness of 5 as the codes of states s_0 and s_3 are not adjacent but $AM_{0,3} = 1$ and $AM_{3,0} = 1$ and the codes of states s_1 and s_2 are not adjacent but $AM_{1,2} = 2$ while the assignments $\{s_0 \equiv 00, s_1 \equiv 11, s_2 \equiv 01, s_3 \equiv 10\}$ has a fitness of 3 as the codes of states s_0 and s_1 are not adjacent but $AM_{0,1} = 1$ and $AM_{,1} = 1$.

The objective of the genetic algorithm is to find the assignment that minimise the fitness function as described in Equation (2). Assignments with fitness 0 satisfy all the adjacency constraints. Such an assignment does not always exist.

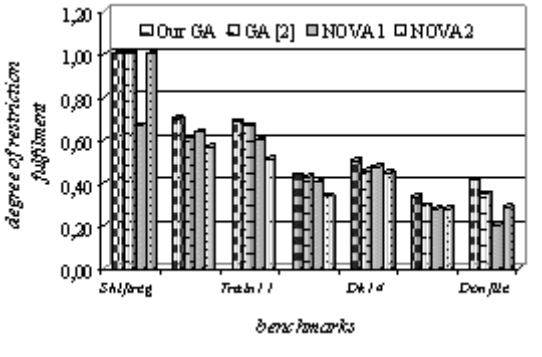

Fig. 4. Graphical comparison of the degree of fulfilment of rule 1 and 2 reached by the systems

Table 2. Best state assignment yield by the compared systems for the benchmarks

FSM	Size	System	State Assignment
Shiftreg	8/16	GA [2]	[0,2,5,7,4,6,1,3]
		NOVA1	[0,4,2,6,3,7,1,5]
		NOVA2	[0,2,4,6,1,3,5,7]
		Our GA	[5,7,4,6,1,3,0,2]
Lion9	9/25	GA [2]	[0,4,12,13,15,1,3,7,5]
		NOVA1	[2,0,4,6,7,5,3,1,11]
		NOVA2	[0,4,12,14,6,11,15,13,7
		Our GA	[10,8,12,9,13,15,7,3,11]
Train11	11/25	GA [2]	[0,8,2,9,13,12,4,7,5,3,1]
		NOVA1	[0,8,2,9,1,10,4,6,5,3,7]
		NOVA2	0,13,11,5,4,7,6,10,14,15,12]
		Our GA	[2,6,1,4,0,14,10,9,8,11,3]
Bbarra	10/60	GA [2]	[0,6,2,14,4,5,13,7,3,1]
		NOVA1	[4,0,2,3,1,13,12,7,6,5]
		NOVA2	[9,0,2,13,3,8,15,5,4,1]
		Our GA	[3,0,8,12,1,9,13,11,10,2]
Dk14	7/56	GA [2]	[0,4,2,1,5,7,3]
		NOVA1	[5,7,1,4,3,2,0]
		NOVA2	[7,2,6,3,0,5,4]
		Our GA	[3,7,1,0,5,6,2]
Bbsse	16/56	GA [2]	[0,4,10,5,12,13,11,14,15,8,9,2,6,7,3,1]
		NOVA1	[12,0,6,1,7,3,5,4,11,10,2,13,9,8,15,14]
		NOVA2	[[2,3,6,15,1,13,7,8,12,4,9,0,5,10,11,14]
		Our GA	[15,14,9,12,1,4,3,7,6,10,2,11,13,0,5,8]
Donfile	24/96	GA [2]	[0,12,9,1,6,7,2,14,11,17,20,23,8,15,10,16,21,19,4,5,22,18,13,3]
		NOVA1	[12,14,13,5,23,7,15,31,10,8,29,25,28,6,3,2,4,0,30,21,9,17,12,1]
		NOVA2	[6,30,11,28,25,19,0,26,1,2,14,10,31,24,27,15,12,8,29,23,13,9,7,3]
		Our GA	[2,18,17,1,29,21,6,22,7,0,4,20,19,3,23,16,9,8,13,5,12,28,25,24]

Table 3. Fitness of best assignments yield by the compared systems

State machine	*#AdjRes*	Our GA	GA [2]	NOVA1	NOVA2
Shiftreg	24	0	0	8	0
Lion9	69	21	27	25	30
Train11	57	18	19	23	28
Bbara	225	127	130	135	149
Dk14	137	68	75	72	76
Bbsse	305	203	215	220	220
Donfile	408	241	267	326	291

5 Comparative Results

In this section, we compare the assignment evolved by our genetic algorithm to those yield by another genetic algorithm [2] and to those obtained using the non-evolutionary assignment system called NOVA [3]. The examples are well-known benchmarks for testing synchronous finite state machines [8]. Table 2 shows the best

state assignment generated by the compared systems. The size column shows the total number of states/transitions of the machine.

Table 3 gives the fitness of the best state assignment produced by our genetic algorithm, the genetic algorithm from [2] and the two versions of NOVA system [3]. The *#AdjRes* stands for the number of expected adjacency restrictions. Each adjacency according to rule 1 is counted twice and that with respect to rule 2 is counted just once. For instance, in the case of the *Shiftreg* state machine, all 24 expected restrictions were fulfilled in the state assignment yielded by the compared systems. However, the state assignment obtained the first version of the NOVA system does not fulfil 8 of the expected adjacency restrictions of the state machine.

The chart of Fig 4 compares graphically the degree of fulfilment of the adjacency restrictions expected in the other state machines used as benchmarks. The chart shows clearly that our genetic algorithm always evolves a better state assignment.

6 Conclusion

In this paper, we exploited evolutionary computation to solve the *NP*-complete problem of state encoding in the design process of asynchronous finite state machines. We compared the state assignment evolved by our genetic algorithm for machine of different sizes evolved to existing systems. Our genetic algorithm always obtains better assignments (see Table 3 of Section 5).

References

1. V.T. Rhyne, *Fundamentals of digital systems design*, Prentice-Hall, Electrical Engineering Series. 1973.
2. J.N. Amaral, K. Tumer and J. Gosh, *Designing genetic algorithms for the State Assignment problem*, IEEE Transactions on Systems Man and Cybernetics, vol., no. 1999.
3. T. Villa and A. Sangiovanni-Vincentelli, *Nova: state assignment of finite state machine for optimal two-level logic implementation*, IEEE Transactions on Computer-Aided Design, vol. 9, pp. 905-924, September 1990.
4. Z. Michalewics, *Genetic algorithms + data structures = evolution program*, Springer-Verlag, USA, third edition, 1996.
5. K. DeJong and W.M. Spears, *Using genetic algorithms to solve NP-complete problems*, Proceedings of the Third International Conference on Genetic Algorithms, pp. 124-132, Morgan Kaufmann, 1989.
6. D.B. Armstrong, *A programmed algorithm for assigning internal codes to sequential machines*, IRE Transactions on Electronic Computers, EC 11, no. 4, pp. 466-472, August 1962.
7. W.S. Humphrey, *Switching circuits with computer applications*, New York: McGraw-Hill, 1958.
8. Collaborative Benchmarking Laboratory, North Carolina State University, http://www.cbl.ncsu.edu/pub/Benchmark_dirs/LGSynth89/fsmexamples/, November 27th, 2003.