Improving Correctness of Finite-State Machine Synthesis from Multiple Partial Input/Output Sequences

Prabhas Chongstitvatana Chatchawit Aporntewan
Department of Computer Engineering, Faculty of Engineering
Chulalongkorn University, Bangkok 10330, Thailand
prabhas@chula.ac.th
Tel: (622)218-6982 Fax: (662)218-6955

Abstract

Our previous work focused on the synthesis of sequential circuits based on a partial input/output sequence. As the behavioural description of the target circuit is not known the correctness of the result can not be verified. This paper proposes a method which increases the correctness percentage of the finite-state machine (FSM) synthesis using multiple partial input/output sequences. The synthesizer is based on Genetic Algorithm. The experimental results show that the correctness percentage can be increased to 100% by increasing of the number of input/output sequences.

1. Introduction

A finite-state machine (FSM) can be constructed from the understanding of its behavior. Each state must be identified to define the state transition function and the output function. Given a behavioral description, the target FSM can be synthesized by many conventional methods.

In contrast, this paper proposes a different approach: an FSM is synthesized not from a behavioural description but from partial input/output sequences. We aim to realize an evolvable hardware that can *mimic* another sequential circuit by observing its partial input/output sequences. This approach is very advantageous to the learning problems in which the internal states are hidden; for example, the FSMs which humans cannot understand their behaviors or FSMs which exist but cannot be easily expressed in the traditional forms (e.g., state diagrams). Today, those problems are beyond the conventional methods.

The FSM consists of state transition table and function mapping input and current state to output. The FSM can be constructed as a lookup table representing state transition and output function. Thus the hardware implementation can be a simple device such as Random Access Memory (RAM). Since the FSM can be directly translated into RAM, it can be seen that the RAM content is evolved.

A method for solving FSM synthesis is based on *Genetic Algorithm (GA)* [5]. GA is a search and optimization algorithm inspired by natural evolution. GA performs search in a *population*, a set of *individuals* which represents points in the search space. At each *generation*, a sequence of genetic operations called *crossover*, *mutation*, and *selection* transforms the existing individuals into a new set of solutions. The solution quality is evaluated in terms of *fitness* in which fitness function must be defined for the problem. The individuals are probabilistically selected to the next generation proportional to their fitness. One necessary condition to improve the solution quality is that the search process does not get stuck at local optima, otherwise all individuals converge quickly to a point. Thus the *diversity* should be maintained in the selection process.

The goal of optimization is to find the fittest individual (FSM) which produces the correct output sequences according to the given input/output sequences. Our previous work [11] shows that the result of synthesis is classified into two categories:

1. **Complete Solution:** a complete solution is a solution that operated correctly for all possible input/output sequences.

2. **Incomplete Solution:** an incomplete solution is a solution that operated correctly for the partial input/output sequences.

In the previous work, the experiment was conducted on single input/output sequence. As the behavioural description of the target circuit is not known the correctness of the solution can not be verified. The correctness percentage is defined as the number of run yielding complete solutions divided by the total number of run yielding solutions. The correctness varied with the length of the input/output sequence. Figure 1 shows the result of a serial adder synthesized from single input/output sequence. Each point on the graph was calculated from 100 runs. The correctness improved with the *length* of the sequence, however, it did not improve further after the le ngth reaches the *upperbound length*.

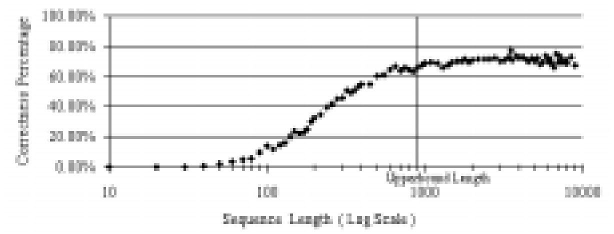

Figure 1. Correctness percentage and sequence length

To improve the correctness further this work proposes using multiple sequences. The main idea is that the performance of a learning system will get better with more examples. The multiple input/output sequences should yield a higher correctness than a single input/output sequence of the same length. The experiment was conducted to synthesis a number of FSMs such as serial adder, modulo counter, reversible counter and sequence detector.

The remaining sections are organized as follows. Section 2 discusses the related works. Section 3 describes the experiment in details. Section 4 presents the experimental results. Section 5 concludes the paper.

2. Related Works

In the early 1960s, L. J. Fogel introduced *Evolutionary Programming (EP)* to predict the next symbol based on a sequence of symbols drawn from finite alphabets [4]. The simulated evolution was performed by modifying a population of FSMs. Five modes of mutation – change an output symbol, change a state transition, add a state, delete a state, or change the internal state – were used to produce new offspring. Typically, μ offspring were produced by mutating each parent [3]. Then, the selection was performed by discarding μ poorest from both parents and offspring. [5, pp. 106] claims that this method is insufficiently powerful due to the lack of structured recombination. In this paper, we prefers to encode FSM into binary string rather than performing genetic operators on the structure of FSM.

The *finite-state automata (FSA)* have been used in the field of *grammatical inference (GI)* which is an instance of the more general problem of *inductive inference* – the task of effectively inferring general rules from examples. The researchers have inferred the general rule by inducing the FSA to accept regular languages [2] or to learn context-free grammars directly from examples [16]. Recently, GA was employed to infer push-down automata from positive and negative samples of unknown languages [10].

The evolvable hardware research [7, 8] presents the synthesis of sequential circuits from random input/output sequence. The GAL16Z8, which is a programmable logic device, represents the sequential circuit. Given an input/output sequence, the configuration bits, also called *architecture bits*, were evolved by means of Genetic Algorithm. It is notable that some solutions were partially correct.

Genetic Algorithm (GA) [5, 9] is used to search for circuits that represent the desired state transition function. The simulated evolution has been used to synthesize finite state machine in [1, 3] where the resulting FSM can predict the output symbol based on the sequence of input symbols observed. In contrast to representing circuits as FSMs, [12] proposes the automated hardware design at the Hardware Description Language level using GA. [6] describes the evolution of hardware at function-level based on reconfigurable logic devices. [13, 14] evolved circuits at the lowest level, in the actual logic devices, using real-time input/output. Our work is similar to [7, 8] in the use of FSM but we use FSM as the model of the desired circuit behaviour.

3 The Experiments

3.1 Genetic Algorithm

The synthesizer is based on GA. We use a small population size and a large number of generation. Our previous experience showed that the diversity of the population must be maintained to prevent pre-mature convergence. Thus, the diversity is maintained in the selection process. The algorithm slightly adapted from [15] is presented as follows:

```
generation = 0;
Initialize P individuals;
While termination conditions not met Do
  Produce Q individuals using crossover;
  Produce R individuals using mutation;
  Select P individuals from (P ∪ Q ∪ R);
  generation = generation + 1;
End While
```

The maximum number of generations was set at 50,000. The P, Q, and R were set at 100, 200, 100 respectively. The genetic operators – crossover, mutation, and selection – are defined as follows:

- **Crossover:** Select a pair of parents randomly from P individuals to produce two offspring using single point crossover.

- **Mutation:** Select a parent randomly from P individuals. Then, mutate it to produce an offspring with $P_m = 0.01$.
- **Selection:** Select best P individuals from $(P \cup Q \cup R)$ individuals to the next generation using combined rank method (fitness rank + diversity rank).

3.2 Encoding Scheme

Each individual represents an FSM by its state transition table. The state transition table is represented by concatenating the next states and the outputs to form a fixed length binary string. Table 1 shows an example, the FSM is encoded by concatenating the next state and the output of all rows together as “00111101”. The length of an individual is determined by the number of state. For real world applications, the number of internal states needed to produced a complete solution might be unknown. We let the number of state of an individual to be larger than the number of state in the target FSM. The solution may contain redundant states and unreachable states. A conventional method can be used to optimize them. The number of available internal states for each circuit is presented in Table 2.

Table 1. Example FSM

State	Input	Next State	Output
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	1

3.3 Input/Output Sequences

The input/output sequences, used in the fitness evaluation, are generated by the following steps:

1. given an target FSM
2. reset the FSM to start state
3. produce a random input sequence
4. feed the input sequence to the FSM and collect the corresponding output sequence
5. repeat steps 2-4 for the next input/output sequences.

3.4 Fitness Function

The fitness of an individual is evaluated by the following steps:

1. fitness, $\mathcal{F}$, = 0
2. reset individual (FSM) to start state
3. feed a given input sequence to the individual to get the corresponding output sequence
4. compare the corresponding output sequence with the given output sequence, $\mathcal{F} = \mathcal{F}$ + *number of similar output bits*
5. repeat steps 2-4 for the remaining input/output sequences

The experiment was conducted on several circuits; serial adder, 1010 detector, 0101 detector, modulo-4 counter, reversible 4-counter, and reversible 8-counter (See Figure 2, 3, 4, 5, 6, and 7 respectively). We ran each case 10 times to produce the correctness percentage in Table 3.

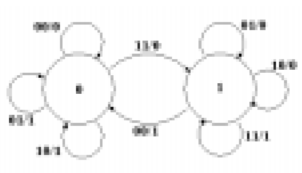

Figure 2. Serial Adder

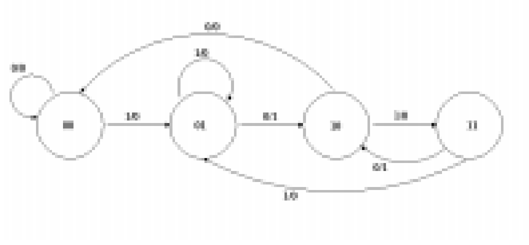

Figure 3. 1010 Detector

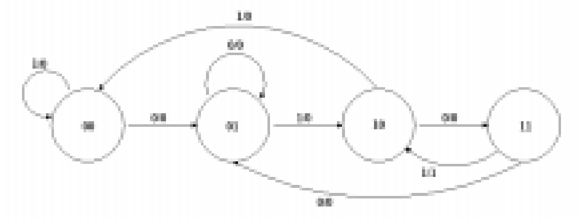

Figure 4. 0101 Detector

Table 2. The number of available internal states

Tested circuits	Input (bits)	Output (bits)	The number of internal states	The number of available internal states
Serial Adder	2	1	2	4
1010 Detector	1	1	4	8
0101 Detector	1	1	4	8
Modulo-4 Counter	1	1	4	8
Reversible 4-Counter	1	2	4	8
Reversible 8-Counter	1	3	8	32

Table 3. Correctness Percentage

Number of Sequences	Correctness Percentage (Sequence Length = 100)					
	Serial Adder	1010 Detector	0101 Detector	Modulo-4 Counter	Reversible 4-Counter	Reversible 8-Counter
1	60.0	0.0	10.0	42.8	20.0	0.00
5	70.0	40.0	10.0	83.6	100.0	57.1
10	80.0	90.0	80.0	100.0	100.0	71.4
25	100.0	55.5	90.0	100.0	100.0	100.0
50	100.0	90.0	100.0	100.0	100.0	100.0
75	100.0	100.0	100.0	100.0	100.0	100.0
100	100.0	100.0	100.0	100.0	100.0	100.0

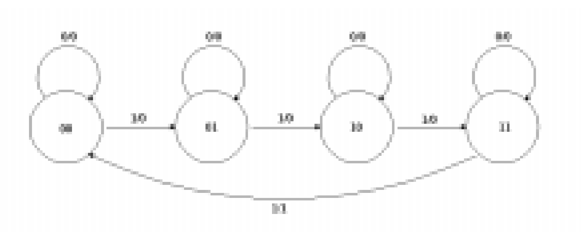

Figure 5. Modulo-4 Counter

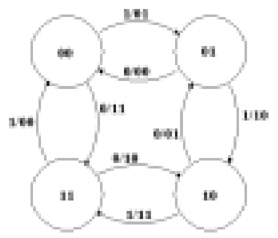

Figure 6. Reversible 4-Counter

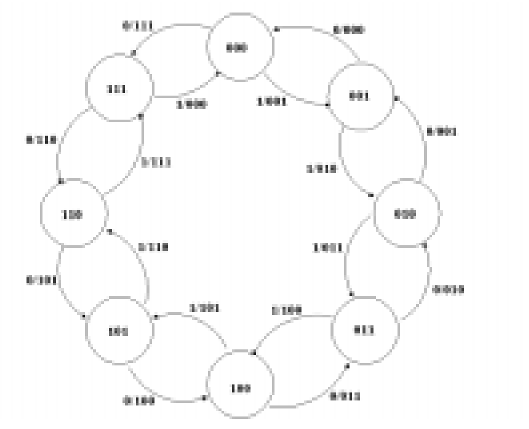

Figure 7. Reversible 8-Counter

4 The Experimental Results

The experimental results in Table 3 show that the correctness is increased with the number of input/output sequence. The correctness can be raised to 100% using 100 input/output sequences. The result of synthesizing a serial adder is analysed in order to understand this improvement. A complete solution is shown is Figure 8. The FSM consisted of a redundant state and an unreachable state; the state “10” is equivalent to the state “11” and the state “01” is unreachable. An incomplete solution produced by using single input/output sequence is shown in Figure 9. The

FSM consisted of 3 parts : the initial state “00”, the part A and the part B. Part A produces incorrect outputs. Part B produces correct outputs. The first few bits in the input sequence determine which part the subsequent states will belong to. Using a single sequence, the FSM that transits to part B will be indistinguishable from a complete solution. Using multiple sequences increase the possibility of excercising part A and hence identify this FSM as an incomplete solution. In other words, the multiple sequence has a better discrimination between complete and incomplete solutions.

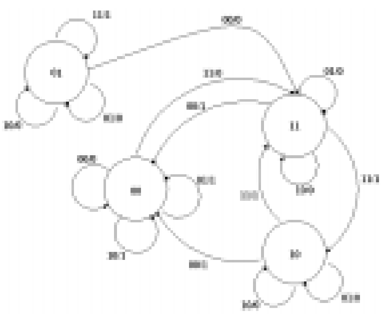

Figure 8. A complete solution (Serial Adder)

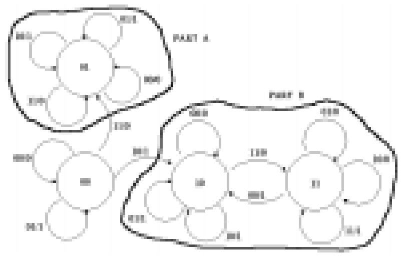

Figure 9. An incomplete solution (Serial Adder)

5 Conclusions

In order to realize an evolvable hardware, we study how to *mimic* another sequential circuit by observing its partial input/output sequences. The evolutionary process has been used to synthesize circuits that perform according to the observed partial input/output sequence. Without the behavioural description of the target circuit, the correctness of the resulting circuit can not be verified. However, the percent of correctness can be increased. The experiment shows that the correctness percentage is increased with the number of input/output sequences. Furthermore, the correctness percentage can be raised to 100%. This has a strong implication on realizing an evolvable hardware in practice. Our future work will concentrate on realizing this idea in the actual hardware.

References

[1] P. J. Angeline, D. B. Fogel, and L. J. Fogel. A comparison of self-adaptation methods for finite state machines in a dynamic environment. In *Proc. of the Fifth Ann. Conf. on Evolutionary Programming*, pages 441–449, 1996.

[2] R. C. Berwick and S. Pilato. Learning syntax by automata induction. *Machine Learning*, 2:39–74, 1987.

[3] L. J. Fogel. Autonomous automata. *Industrial Research*, 4:14-19, 1962.

[4] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons, New York, 1966.

[5] D. E. Goldberg. *Genetic Algorithm in search, optimization and machine learning*. Addison-Wesley, 1989.

[6] T. Higuchi, M. Murakawa, M. Iwata, I. Kajitani, W. Liu, and M. Salami. Evolvable hardware at function level. In *Proc. of Int. Conf. on Evolutionary Computation (ICEC’97)*, pages 187–192, 1997.

[7] T. Higuchi, T. Niwa, T. Tanaka, H. Iba, H. de Garis, and T. Furuya. Evolving hardware with genetic learning: A first step towards building a darwin machine. In *Proc. of Int. Conf. on Simulation of Adaptive Behavior (SAB’92)*, 1992.

[8] T. Higuchi, T. Niwa, T. Tanaka, H. Iba, and T. Furuya. A parallel architecture for genetic based evolvable hardware. In *Proc. of Int. Joint Conf. on Artificial Intelligence (IJCAI’93), Workshop on Parallel Processing for Artificial Intelligence*, pages 46–52, 1993.

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

[10] M. M. Lankhorst. A genetic algorithms for the induction of pushdown automata. In *Proc. of Int. Conf. on Evolutionary Computation (ICEC’95)*, pages Vol. 2, 741–746, 1995.

[11] C. Manovit, C. Aporntewan, and P. Chongstitvatana. Synthesis of synchrounous sequential logic circuits from partial input/output sequence. In *Proc. of Int. Conf. on Evolvable Systems (ICES’98)*, pages 98–105, 1998.

[12] J. Mizoguchi, H. Hemmi, and K. Shimohara. Production genetic algorithms for automated hardware design through an evolutionary process. In *Proc. of Int. Conf. on Evolutionary Computation (ICEC’94)*, pages 661–664, 1994.

[13] A. Thompson. An evolved circuit, intrinsic in silicon, entwined with physics. In *Proc. of Int. Conf. on Evolvable Systems (ICES’96)*, pages 390–405, 1996.

[14] A. Thompson. The natural way to evolve hardware. In *Proc. of IEEE International Symposium on Circuits and Systems (ISCAS’96)*, pages Vol. 4, 37–40, 1996.

[15] P. H. Winston. *Artificial Intelligence*, pages 505–528. Addison-Wesley, 1992.

[16] H. Zhou and J. J. Grefenstette. Induction of finite automata by genetic algorithms. In *Proc. of Int. Conf. on Systems, Man and Cybernetics*, pages 170–174, 1986.