Improving FSM Evolution with Progressive Fitness Functions

Jason W. Horihan*and Yung-Hsiang Lu
School of Electrical and Computer Engineering, Purdue University
horihan@purdue.edu and yunglu@purdue.edu

ABSTRACT

This paper presents a method to reduce the total number of generations needed to evolve a finite state machine using genetic inferencing. Genetic inferencing is an evolution method that creates designs from their input-output relationship. We reduce the time required to evolve a design by only evolving a small partition of the input-output relationship. We construct the complete design by repeatedly evolving different input-output relationship partitions. Genetically inferring a design with our method can reduce evolution time by more than 80%.

Categories and Subject Descriptors

B.1.2 [**Control Structures and Microprogramming**]: Control Structure Performance Analysis and Design Aids—*Automatic synthesis*; J.6 [**Computer-Aided Engineering**]: Computer-aided design(CAD)

General Terms

Automated Design

Keywords

Evolution, Evolutionary Design, FSM, Finite State Machine, Genetic Inference, Fitness, Fitness Function

1. INTRODUCTION

The basic concept of evolution is to mold a group of candidates, or *population*, into a set that satisfies user-defined constraints. With a computer simulation, we can evolve finite state machines to provide desired functionality. We call this procedure Evolutionary Computing (EC). EC finds better designs by iteratively applying evolutionary operators such as selection, combination, and mutation to candidates. In this way, EC represents a search strategy that finds the best solution based on a given set of criteria.

EC has also been previously used for optimization problems. Optimization begins with a group of solutions that already operate correctly. EC attempts to improve the quality of the solutions. The *fitness* is a function that associates a scalar quantity with the quality of a design. The more optimized a design is, the higher fitness it receives. A solution is improved if it correctness is retained while the fitness increases. EC improves the population with respect to the fitness function. Our research focuses less on optimization and more on the evolution of correct design functionality. This paper focuses on creating finite state machines (FSM) because they are important components of complex systems. In particular, we evolve FSMs to recognize regular expressions. Regular expressions are used to represent a subset of strings, therefore suited well to genetic inferencing. FSM evolution can be beneficial because it requires no human interaction. It can be used for data analysis, pattern matching or profiling. It can also be used in an adaptive system. The FSM would adapt to its environment to improve its fitness level.

We use *genetic inference* [4] to evolve FSMs that represent regular expressions match a given subset of input strings Genetic inference is an evolutionary method where an implementation is created for a given input and output specification. The population "infers" what the solution should be. In our approach, groups of strings from a given set are used to create regular expressions. Candidates attempt to match each string in a group. The candidate that correctly match the most strings have a high fitness and are selected to remain in the next generation. Genetic inference evolves a design with no knowledge of the target design. It begins with randomly assembled FSMs that may or may not match any of the provided input strings. The fitness of each candidate is assessed by attempting to match or reject each string. Matching and rejecting the correct strings gives a better fitness, and hence, a better design.

The contribution of this work is a new method of the genetic inference evolution strategy. Our method reduces the number of generations required to evolve FSMs to recognize regular expressions. We design fitness functions for small increments during evolution. For example, we may begin by evolving a regular expression that matches "aa". When a

*Jason W. Horihan is now with Intel. This work is supported in part by Altera.

GLSVLSI'04, April 26–28, 2004, Boston, Massachusetts, USA.

solution is reached, we change the target functionality to be slightly more complex such as "aab" and continue evolving. The target is changed several times until the full functionality of the design is achieved. Our method demonstrates significant speedup for regular expressions of different sizes.

2. RELATED WORK

Artificial evolution has been applied to several circuit design or optimization applications. Miller et al. [8] evolved gate-level digital circuits and created 2-bit adders and 3-bit multipliers with fewer gates than human-designed counterparts. Erba et al.[6] used evolution to design low-power digital filters. Drechsler [5] evolved the solutions for physical designs and Valenzuela et al. [12] optimized placement and area for VLSI circuits . Harvey el al. [7] used an FPGA to evolve an asynchronous tone discriminator. The programming data of the FPGA was used as the design, with combination and mutation done on the individual FPGA programming bits. Koza et al. [9] also used an FPGA to evolve a minimal n-step 7/8/9 sorter. Each sorting network in the population was converted to an FPGA programming file, configured, and evaluated. Amaral et al. [1] applied evolution to the state assignment problem of FSMs. Chellapilla et al [3] evolved modular FSMs; a main FSM was evolved in parallel with sub-FSMs. Control would pass between the main FSM and the sub-FSMs. Chongstitvatana et al. [4] used genetic inferencing to synthesize FSMs using multiple input/output sequences. Brave [2] applied cellular encoding to aid the evolution of language recognition FSMs. Nippaman et al. [11] provided another FSM inference method, one where only the state transitions were evolved. State outputs were determined post-evolution. Lankhorst demonstrated that evolution could be used to infer pushdown autonoma from correct and incorrect examples of a language [10].

Our method differs from [2] [4] [10] [11] by using "incremental" inference. Previous methods evolve solutions directly by trying to infer full functionality from the start of evolution. Direct evolution suffers from local optimums. A local optimum is a design with a higher fitness than similar designs. In this case, evolution may have a difficult time "moving on" since the similar designs have lower levels of fitness. Thus, evolution becomes trapped at this local optimum. Our method evolves designs in small increments. Small increments reduce the number of local maximums by limiting the range of the search space.

3. EVOLUTION OF FSM

An FSM reads an input string from a regular expression and accepts or rejects the string. A regular expression is created by the following rules: (1) a finite set of alphabets, (2) union, intersection, or concatenation of two regular expressions, (3) complement of a regular expression, (4) repetition of a regular expression, also called Kleene closure and (5) the empty string. We evolve regular expressions because (1) the solutions generated are easy to validate and (2) different regular expressions are easily available to evolve.

3.1 Regular Expression Fitness

A typical FSM has only two outputs: "accept" and "not-accept". Hence, it is difficult to compare the degree of correctness of two FSMs. We can only determine if the FSM was correct or incorrect, not how close it was to being correct. Our solution is two-fold. First, we expand the number of outputs during evaluation. We check the output after reading every single input character by assuming that each character is the final character of the input string. The associated final output of the regular expression is then the output of the FSM. Second, we divide the "not-accept" output into two separate outputs: "reject" and "unknown". The "unknown" output is the output of the FSM while parsing is in progress. If a character is read that causes parsing to fail, the output is "reject". Control then stays in the "reject" state because no further inputs could create a match.

For example, figure 1 shows the parsing of a string using our method. Figure 1(a) shows a candidate for the target regular expression "(aba)*". All "reject" states are removed for clarity. Notice that the candidate is correct except for the output '1' of one node. A valid test, "abaaba", is parsed through the candidate. Its output is shown in figure 1(b). The intended output is shown in figure 1(c). The candidate and intended target output strings are compared and fitness is based on the number of correct comparisons.

3.2 Mutation and Crossover

Mutation of each candidate is completed by one of five independent graph operations: add a state, delete a state, reassign the start state, reassign an edge, and reassign an output. Crossover is the process of combining two finite state machines into a new one, with components of both contained in the new FSM. An important concept of crossover is that characteristics of each parent FSM are retained in the new FSM. Parents are randomly divided (Figure 2) Parts from each parent are combined to form a new FSM.

3.3 Incremental Evolution

To understand why evolving incrementally is generally faster, we consider two import aspects of evolution. First, we consider combination and mutation. Combination makes a major change to an FSM, while mutation makes a much smaller modification to an FSM. This means that combination can make a design move rapidly over the design space while mutation moves through the design space in much smaller increments. Combination moves across a large distance in the design space and quickly discovers a solution. Mutation refines a design by individual tweaking with little design space movement.

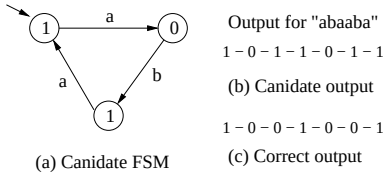

Figure 1: The Outputs of an FSM

Second, evolving population must travel though the design space to achieve the target design. There may be several paths through the design space that evolution can follow. Each path is generally defined by the fitness function. The population's position in the design space will move toward a position of higher fitness. This movement outlines a path through the design space ending in the target design. If there are multiple paths toward the target, some candidates of the population may follow different paths. As long as the fitness levels of designs on each path are comparable, evolution will proceed down each path.

The problem with direct evolution presents itself. The population is split along several design paths. During the combination process, randomly selected parents will combine and form new FSMs. If two parents are chosen from different (and dissimilar) evolution paths, their child FSM may not fall within a evolution path. Its fitness may be much lower than the rest of the population and it is not selected to be passed on. The effectiveness of the combination operator is reduced and evolution is achieved by the slower mutation method.

Our method improves direct evolution by using small increments. Evolving in small increments requires less design space movement. Small amounts of functionality are added on each iteration. We also know that since only a small amount of functionality is added, the new target is not far from the previous target in the design space. Fewer evolution paths exist to the intermediate target. The combination operator has a greater number of compatible designs to combine, so incrementally evolving improves the speed of combination.

For example, instead of directly evolving "aababa", we would first evolve "aa". When the solution to "aa" is available in the population, the evolution target is changed to "aaba". Evolution continues until a solution is found, when the target is changed to the final target, "aababa". Section 4 shows various results of step evolution with different step sizes and different evolutionary targets.

4. EXPERIMENTS

Our experiments compare direct evolution with incremental evolution using several incremental sizes. In each experiment, we evolve a regular expression over several trials. Most test cases were able to finish. The case "aab(ba)*bba(ab|ba)bb", however, was too complex for direct evolution, while incremental evolution was able to create a correct solution. The population size is 300 FSMs. The makeup of the next generation is 1/3 from selection, 1/3 from combination, and 1/3 from mutation. Genetic Inference requires multiple tests, so the number of tests is set to 500. The maximum number of generations allowed are 10 thousand. After a candidate becomes functionally correct, the population continues to evolve for 50 generations to optimize the FSM size.

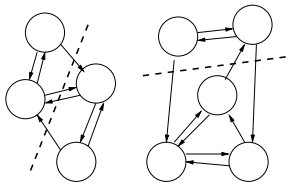

Figure 2: Partitioned Parent Graphs

4.1 String Evolution

The experiment evolve regular expressions matching various lengths of strings. For comparison purposes, we use the median value of all test groups. In the first case, "abba" was evolved. It was evolved with 0 character increments (directly), with one character increment ("a", "ab", "abb","abba") and with two character increment ("ab","abba"). Figure 3 shows the results of this test. In the graph, the vertical axis is the median number of generations needed to evolve each test. Each bar in the graph represents a string increment size. The '0' bar represents direct evolution, while the other bars indicate the character increment size. Evolving incrementally results in a reduction in the number of required generations by up to 16%. An interesting observation is that even with four fitness changes, with each fitness change having a state optimization phase, incremental evolution still evolved in fewer generations than the direct method. The benefits of incremental evolution

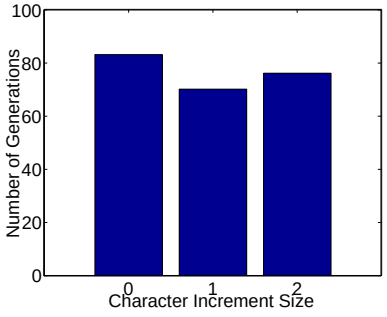

Figure 3: Four Character String Evolution

on strings evolution were more pronounced when evolving longer strings. Figures 4 and 5 show the results of evolving strings of 6 and 12 characters. Evolving a length of 6 incrementally by 3 characters reduces the number of generations needed by 50%. Evolving a length of 12 by 1 character increments gives us a generation reduction of 80%. We see that both the one and three character increments evolve in fewer generations. The reduction in the three character increment can be explained by observing the target string. In these tests we attempted to evolve "abbabb" and "abbabbbabaab". We see that the first three characters "abb" are repeated in the target string. Once the "abb" regular expression is evolved, it is readily available to be used in later FSMs. Evolving from "abb" to "abbabb" is a short step.

4.2 Union Evolution

Our tests were done on 3 character unions. The first evolved a set of six unions: "aaa", "aab", ... "bab". The second test evolved the first set with an additional "bba" string. The first test evolved in few generations, with the direct evolution method the fastest. Figure 6 shows the re-

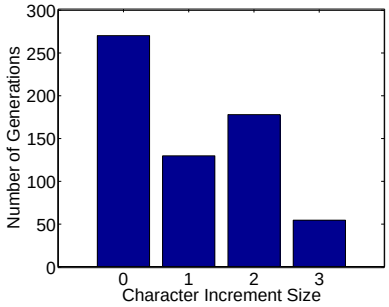

Figure 4: Six Character String Evolution

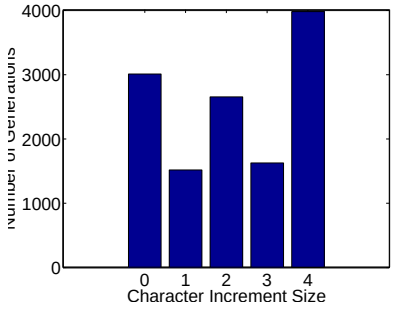

Figure 5: Twelve Character String Evolution

sults. Notice that evolution by increments of two and three did not fall far behind the direct method. Incremental evolution did not work well with one literal increments. Incremental reinforcement was not available. For example, when evolving "aaa", we also reject "aab", etc. After moving the target to "aaa|aab", evolution must remove the components of "aab" rejection and add the correct functionality. The

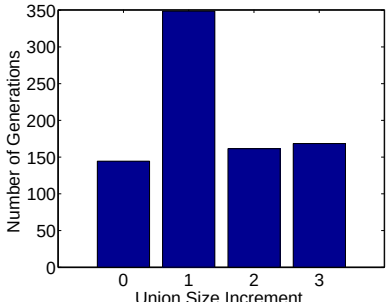

Figure 6: Six Literal Union

second test evolved with significantly more generations. Excluding only the "bbb" component required a more complex FSM. The added complexity slowed the evolution of the direct method. Figure 7 shows the results of the test, with evolving by two literals reducing the required number of generations by 88%. Notice that almost all incremental test performed significantly better than the direct test.

5. CONCLUSION

Regular expressions are evolvable by EC with minor adjustments. Genetic inferencing is a evolutionary method to

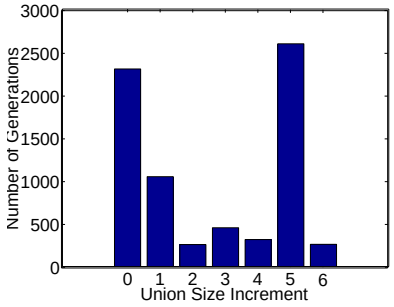

Figure 7: Seven Literal Union

create an implementation of an input-output relationship. We show that this method is enhanced by the addition of incremental fitness targets. By carefully selecting a path of evolution targets, designs can evolve in much fewer generations.

6. REFERENCES

[1] J. Amaral, K. Turner, and J. Ghosh. Designing Genetic Algorithms for the State Assignment Problem. *IEEE Transactions on Systems, Man, and Cybernetics*, 686–694, 1995.

[2] S. Brave. Evolving deterministic finite automata using cellular encoding. In *Genetic Programming*, pages 39–44 1996.

[3] K. Chellapilla and D. Czarnecki. A Preliminary Investigation into Evolving Modular Finite State Machines. In *Congress on Evolutionary Computation*, pages 1349–1356, 1999.

[4] P. Chongstitvatana and C. Aporntewan. Improving Correctness of Finite-State Machine Synthesis from Multiple Partial Input/Output Sequences. In *Workshop on Evolvable Hardware*, 1999.

[5] R. Drechsler. *Evolutionary Algorithms for VLSI CAD*. Kluwer, 1998.

[6] M. Erba, R. Rossi, V. Liberali, and A. G. Tettamanzi. An Evolutionary Approach To Automatic Generation of VHDL Code for Low-Power Digital Filter. In *European Conference on Genetic Programming*, pages 36–50, 2001.

[7] I. Harvey and A. Thompson. Through the Labyrinth Evolution Finds a Way: A Silicon Ridge. In *International Conference on Evolvable Systems*, pages 406–422, 1996.

[8] J F Miller et al. Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study. In *Genetic Algorithms and Evolution Strategy in Engineering and Computer Science*, pages 105–131, 1998.

[9] J R Koza et al. Evolving Computer Programs using Rapidly Reconfigurable FPGAs and Genetic Programming. In *International Symposium on Field Programmable Gate Arrays*, pages 209–219, 22-24 1998.

[10] M. Lankhorst. A Genetic Algorithm for Induction of Nondeterministic Pushdown Automata. Technical report, University of Groningen, Computer Science, CS-R 9502.

[11] N. Nipaman and P. Chongstivatana. An Improved Genetic Algorithm for The Inference of Finite State Machine. In *IEEE International Conference on Systems, Man and Cybernetics*, pages 1–5, 2002.

[12] C. L. Valenzuela and P. Y. Wang. VLSI Placement and Area Optimization Using A Genetic Algorithm To Breed Normalized Postfix Expressions. *IEEE Transactions on Evolutionary Computation*, 390–401, August 2002.