Genetic Programming and its Applications to the Synthesis of Digital Logic

Karen M. Dill, James H. Herzog*, and Marek A. Perkowski

Portland State University
Department of Electrical Engineering
Portland, Oregon 97207
e-mail: dills@worldnet.att.net / mperkows@ee.pdx.edu
Tel: 503-725-5411 / Fax: 503-725-4882

*Oregon State University
Department of Electrical and Computer Engineering
Corvallis, Oregon 97331
herzog@ece.orst.edu
Tel: 541-737-2973 / Fax: 541-737-1300

Abstract: In this research Genetic Programming is applied to the synthesis of arbitrary logic expressions. As a new method of logic synthesis, this technique is uniquely advantageous in its flexibility for both problem applicability and optimization criterion. A number of experiments were conducted exploring this method with different types of logic gates and population sizes. While complete function coverage is not guaranteed, the best experimental test results over eight randomly designed functions, of four to seven input variables, have produced logic equations with a 98.4% function coverage. In addition, the relation between the training set size for the Genetic Program and function coverage was also empirically explored. These experiments showed that only small training sets were necessary for function recognition.

I. INTRODUCTION

Natural systems are extremely well adapted to their environments, as living organisms demonstrate the propensity to solve a multitude of complex problems for both survival and growth, through instinctual, experiential, and intellectual means. As nature has proven its robust system design, the biologically inspired evolutionary process is here applied to the synthesis of digital hardware circuits.

Genetic Programming (GP) is a means of developing functions through the evolutionary process of natural selection. Through a process of emergent intelligence, the GP formulates solutions based on accumulated knowledge and the merit of potential solutions. In recent years genetic operators, have been successfully applied to a wide range of engineering problems. However, with the realization of computer designed algorithms, very little research to date has applied these concepts to digital logic.

An original automated logic design technique of evolutionary "Genetic Hardware" is herein developed. In this method, the logic function to be synthesized must be specified by a truth table. The input signals and defined gates for the logic are then automatically encoded as binary strings. Resulting from the evolutionary process, "breeding" and evaluating a number of designs over "generations", the synthesized logic function is derived.

As a new method of logic synthesis, this technique offers unique advantages. The logic is designed by evolutionary means without explicit computer programming, making the process "hands-off" for the user. The technique is applicable to solving a large number of logic design problems, regardless of function size and type. It is also a multipurpose design method, offering great flexibility. In contrast to other means of logic synthesis, this new method of circuit design can be completely customized to optimize for virtually any cost function, i.e. circuit area, power, delay, number of gates, speed, etc. The desired optimization goals and their relative importance need only be described by a numerical value of merit. Currently implemented only for the correct functionality of Boolean logic with a minimum number of gates, the method is adaptable for multi-valued logic. Additionally, any complex gates described by truth tables can be utilized. As the evolutionary mechanisms remain unchanged, there is no differentiation of problem difficulty based on these criteria.

With genetic hardware development, the circuit implementation is often different in appearance from that of a traditionally designed circuit. Since design verification is already incorporated into the circuit development algorithm, it is assured that, if the 100% coverage is achieved, the circuit will function according to specification, so that no test generations or simulations are necessary to verify its correctness.

II. GP APPLICATION TO LOGIC SYNTHESIS

Genetic Programming utilizes the process of natural selection, i.e. survival of the fittest over a period of time, to develop robust computer code, representative of functions. Using this technique, hierarchical data structures are encoded as binary strings, analogous to biological chromosomes, to which the standard operations of a Genetic Algorithm are applied. These operations are modeled after natural processes and include reproduction, crossover, and mutation. As the structure of the Genetic Program has been implemented with a "rooted, point-labeled tree with ordered branches", this provides a generic model for any function or program. Within this model, data structures are divided into the categories of "functions" and "terminals". Conventionally the GP paradigm has been applied to the evolution of LISP code. However, redefining "functions" as logic gates and "terminals" as binary or multi-valued logic signals, the data structure can be directly applied to the evolution of digital logic equations.

The genetic programming code serves as a framework for the artificial evolutionary process. The foundation of the software code used for this research is a public domain genetic programming system. It is available via anonymous ftp to ftp.cc.utexas.edu in the /pub/genetic-programming/code directory. The particular code which was chosen, "Genetic Programming in C++, Gpc++ Version 0.40" was written in C++ by Adam Fraser, at the Department of Electronic & Electrical

0-7803-3905-3/97/$10.00

Engineering - Cybernetics Research Institute, University of Salford, Salford, UK. This software is a very general evolutionary workhorse and was adapted for the GP-Logic Synthesis problem.

III. EXPERIMENTAL RESULTS

The GP-Logic Synthesis was applied to a number of randomly selected 4, 5, 6, and 7 variable binary equations as a means of obtaining empirical results regarding optimal run parameters for digital design. Parameters that varied in this study included the types of logic gates, population sizes, the number of input variables, and the percentage of truth table "don't care" outputs.

For the experiments in this research, the fitness function employed was simply a count of the number of correct function outputs, which was then directly comparable to the total function outputs. In actuality, this "counting" of terms and gates is only a very rough measure of the circuit minimization. The real cost and quality of a logic synthesis design depends on its matching with the technology of implementation (PLA, FPGA, etc.). The fitness could easily be calculated as any value of merit, thus optimizing for circuit speed, layout area, power consumption, or testability, in any combination and priority.

The following empirical experiments are an exploration of the logic synthesis design method proposed. These trials serve both as an initial experimental indicator of the relation between GP performance and the control variables in the artificial evolution system, as well as the merit of this new method of logic synthesis.

A. Experimental Run Parameters and Logic Functions

There are a number of run parameters for the GP-Logic Synthesis which were constant for all of the experimentation. The creation type utilized for the GP was Ramped Half and Half: half the population is created with an even distribution of chromosomes ranging in length from the minimum to maximum depth allowed and the other half of the population contains chromosomes of the maximum depth only. The maximum tree depth at creation was 6. The maximum depth at crossover was 17. Details regarding the run parameters are available in the software documentation, cited in the paper reference section.

The experiments were conducted on randomly composed 4, 5, 6, and 7 variable functions, hereafter labeled and defined as follows.

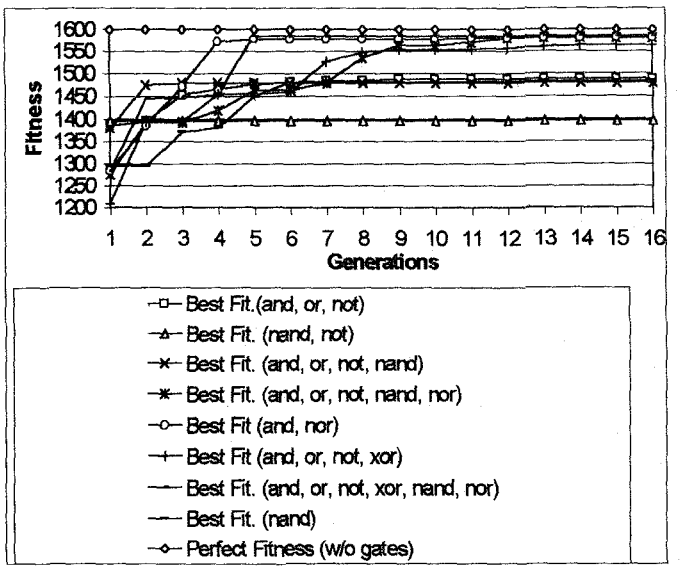
Test 1: $f(a,b,c,d) = \Sigma(0,4,5,7,8,9,13,15)$
Test 2: $f(a,b,c,d) = \Sigma(4,6,7,15)$
Test 3: $f(a,b,c,d,e) = \Sigma(5,6,9,10)$
Test 4: $f(a,b,c,d,e) = \Sigma(1,2,6,7,9,13,14,15,17,22,23,25,29,30,31)$
Test 5: $f(a,b,c,d,e,f) = \Sigma(1,7,11,21,30)$
Test 6: $f(a,b,c,d,e,f) = \Sigma(10,12,14,20,21,22,25,33,36,45,55)$
Test 7: $f(a,b,c,d,e,f,g) = \Sigma(20,28,52,60)$
Test 8: $f(a,b,c,d,e,f,g) = \Sigma(20,28,38,39,52,60,102,103,127)$

B. Types of Logic Gates

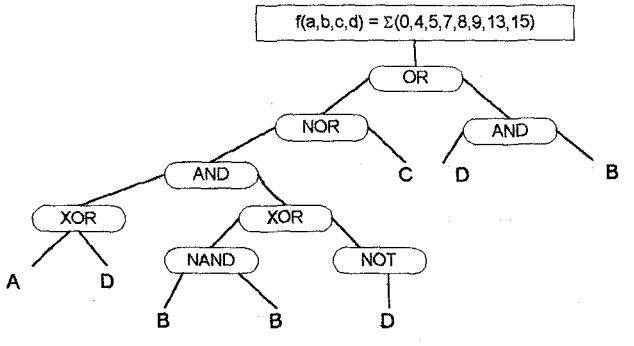
To empirically determine which types of logic gates are most effective for GP-Logic Synthesis, tests were conducted with groups of gates with logical closure. All experiments utilized two-input gates (with the exception of NOT). The investigation was conducted as follows.

Population Size:	1000
Mutation Prob. Rate:	1/1000
Function Set: (all are 2-input gates, except the 1-input NOT)	Case 1: and, or, not Case 2: nand, not Case 3: and, or, not, nand Case 4: and, or, not, nand, nor Case 5: and, nor Case 6: and, or, not, xor Case 7: and, or, not, xor, nand, nor Case 8: nand
Fitness Measure:	+100 for each correct minterm -1 for each logic gate/terminal
Criterion:	Fitness is ((2^n) - gates/terminals)*100 Where n is the number of input variables
Termination:	50 generations

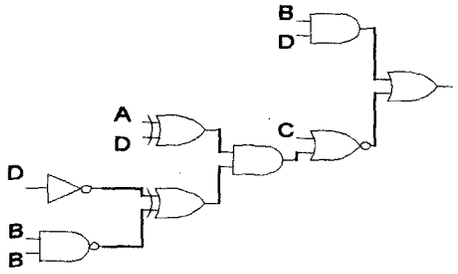
A representative graph showing the evolution of the synthesized logic is shown for the Test 1 problem.



The GP tree structure for the logic gate Case 7 solution to the Test 1 problem, is the following.



The schematic diagram for this synthesized circuit is as follows.



The circuit can be simplified with other programs, such that redundant logic is eliminated. For example the "B NAND B" shown here would simplify to "NOT B", as well as the logic "NOT B XOR NOT D" could be changed to "B XOR D". This then, creates a practical and good, but not standard solution. This structure is quite different than that of the human design, in SOP Form.

The best function coverage results, for each logic gate case group, are shown in the following table. The number of gates and terms in the evolved logic function are given for the derived logic equations.

Test	Correct/ Total Minterms	Cover	Gates/Terms in Function Logic Gates Case: 1	2	3	4	5	6	7	8
1	16/16	100.0%	X	X	X	19	19	29	16	X
2	16/16	100.0%	11	18	12	15	13	10	10	19
3	32/32	100.0%	27	X	29	X	X	18	9	X
4	31/32	96.9%	X	17	17	20	X	14	13	X
5	62/64	96.9%	X	X	X	X	X	X	*16	87
6	58/64	90.6%	X	X	23	X	X	26	X	X
7	128/128	100.0%	X	X	10	X	X	X	13	X
8	126/128	98.4%	X	X	X	X	X	X	X	63

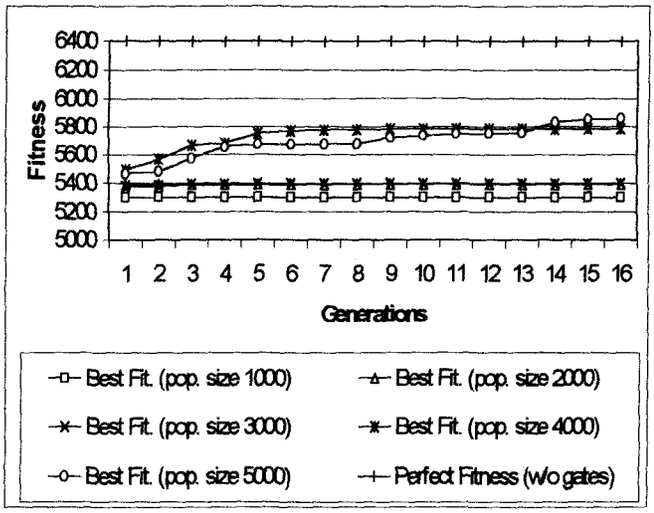
*61/64 mintermss correct
X = Only best function coverage results listed in table.

C. Population Size

The GP Logic Synthesis experiments were next conducted using different sizes of string populations. The purpose of this inquiry was to get a basic understanding of the impact that larger populations, by introducing a greater breeding selection and perhaps larger diversity into the gene pool, would have on function discovery. The complete set of logic gates (AND, OR, NOT, XOR, NAND, NOR) were utilized.

Population Sizes:	1000, 2000, 3000, 4000,5000
Mutation Rate:	1/1000 (probability)
Function Set:	and, or, not, xor, nand, nor
Fitness Measure:	+100 for each correct truth-table minterm -1 for each logic gate/terminal
Criterion:	Fitness is ((2^n) - gates/terminals)*100 Where n is the number of input variables
Termination:	50 generations

A representative graph detailing the evolution of the synthesized logic is shown for the Test 6 problem. None of the experiments produced a solution with 100% coverage.



The study was conducted only on those arbitrarily chosen functions for which incomplete solutions (function coverage of less than 100%) were obtained in the preceding logic gate study. These results are given, as follows.

Test	Correct/ Total Minterms	Cover	Population Sizes: 1000	2000	3000	4000	5000
4	32/32	100.0%	X	X	31	30	13
5	62/64	96.9%	X	X	18	19	X
6	59/64	92.2%	X	X	X	X	27
8	124/128	96.9%	X	X	X	X	21

X = Only best coverage results listed in table

D. GP Training Set Size

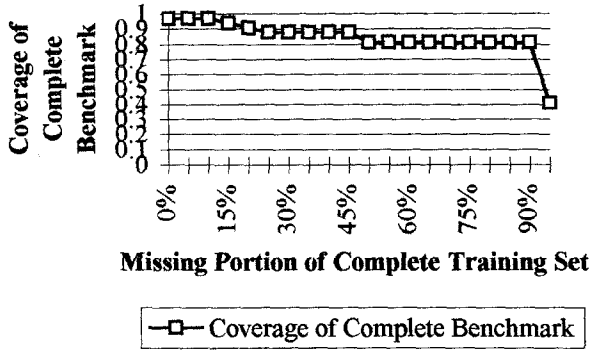
Empirical experiments were also conducted to understand the relation between GP training set size and function coverage with the new logic synthesis method. Testing was performed on the MCNC benchmarks "9sym" and "Majority". 9sym is a completely defined, 9-input, 1-output function of $2^9 = 512$ test vectors, with symmetric properties. The "Majority" function is a 5-input, 1-output function, which is also a well-defined standard. The third test was defined in the previous section as Test 6. With 6-inputs and 1-output, it is slightly larger than the "Majority" benchmark, but not so large as the "9sym" file. As this function design is completely random, it does not demonstrate the "pattern-ness" or regularity most often observed in real-world engineering problems.

For each of the test functions the experimentation was conducted as follows. In the first iteration of the experiments, the GP-Logic Synthesis was trained over the benchmark's truth table, with a 0% missing portion (don't cares) of the complete training set. (This test was identical in manner to that in all the previous experiments.) Subsequent tests were then conducted in which the GP was only provided a portion (arbitrarily missing

from 5 to 100%) of the benchmark truth table during the training session, run for a period of 100 generations. The GP derived function that was found to provide the best coverage of the training set, at the end of the final generation (100) was selected. The logic outputs produced by the synthesized equation, over the entire range of inputs, were then compared to those of the benchmark's truth table. This provided a measure of the complete function coverage in relation to the training set.

Results are given for the Majority benchmark. It was observed that for training sets missing from 0 to 90% (don't cares) of the complete benchmark, (or conversely, training sets that consisted of from 100 to 10% of the benchmark), that complete function coverage was 80% or greater.

Majority: Training Set Size vs. Complete Function Coverage



Similar behavior in the relation between training set size and complete function coverage was observed in all three experiments. These results are given. (Note that 100% don't cares indicates complete random number generation.)

Test	Coverage of Complete Benchmark										
	Percent don't cares										
	0	10	20	30	40	50	60	70	80	90	100
9sym	87%	89%	86%	85%	84%	84%	86%	85%	81%	69%	50%
Maj.	97%	97%	91%	88%	88%	81%	81%	81%	81%	81%	41%
Test 6	86%	84%	81%	81%	81%	81%	69%	44%	44%	47%	55%

IV. GENERAL OBSERVATIONS

From the preceding experiments a number of observations were made. In general, the Genetic Programming demonstrates certain trends when applied to the logic synthesis problem. The GP usually tends to do better (hereafter defined as achieving a higher function coverage) when there is a larger choice of logic gates. In this circumstance it can better choose the gate that meets a specific need by either more efficiency (requiring fewer gates) or by effecting a particular part of the logic, leading to more correct outputs. (An exception to this generality is sometimes observed in the experiments implementing the synthesis with only NAND gates. Such synthesized equations can achieve good function coverage, but often have high structural complexities. The NAND gate was the only gate tested, that is universal by itself.) The GP also does better with larger sized populations. This provides a bigger pool of chromosomes from which the best traits can be selected. In the "Don't Cares" versus Function Coverage experiments, it was observed that only a small training set is necessary for function recognition. The number of experiments in this research was quite small; these results may be highly biased by the degree of "pattern-ness" inherent in these functions. However since most natural functions do exhibit a high degree of pattern, this may not be an issue.

This empirical experimental evidence is a good, practical, and effective manner for an understanding of the effectiveness of GP for the application of logic synthesis. It is a first step to an implementation of a more comprehensive logic synthesis method and more testing on standard benchmarks is necessary.

V. CONCLUSION

A new method of logic synthesis utilizing Genetic Programming was created. In this method, the logic is designed entirely by evolutionary means, without explicit computer programming. This technique is uniquely advantageous for its flexibility, both in its applicability to any type of logic (binary or multi-valued), and its optimization criterion (correct logic outputs, minimal layout area, minimal power consumption, etc.) In addition, it was found that only small training sets are necessary for the evolutionary function discovery. While complete function coverage is not guaranteed, the best experimental test results over eight randomly designed functions have produced logic equations with a 98.4% function coverage. Research endeavors continue towards proven coverage by analyzing the differences between the derived logic and given truth table. However, this technique is already useful for the Constructive Induction Approach to Machine Learning, where the synthesis goal is to minimize the classification error, rather than attain complete coverage. In Machine Learning, finding a simpler circuit without full coverage is a way of dealing with the "over-fitting" problem.

REFERENCES:

[1] K. M. Dill, *Growing Digital Circuits: Logic Synthesis and Minimization with Genetic Operators*, Master of Science Thesis, Dept. of Electrical and Computer Engineering, Oregon State University, June 1997.

[2] A. Fraser, "Genetic Programming in C++: (A manual in progress for gpc++, a public domain genetic programming system)". *Technical Report #040.* University of Salford, Cybernetics Research Institute, 1994.

[3] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, & Machine Learning*, New York: Addison-Wesley Publishing Company, Inc., 1989.

[4] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Cambridge, Massachusetts: The MIT Press, 1992.

[5] J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*, Cambridge, Massachusetts: The MIT Press, 1994.