http://dx.doi.org/10.5755/j01.eee.19.7.5169 ELEKTRONIKA IR ELEKTROTECHNIKA, ISSN 1392-1215, VOL. 19, NO. 7, 2013

Optimization of Combinational Logic Circuits with Genetic Programming

S. Karakatic[1], V. Podgorelec[1], M. Hericko[1]
[1]*Institute of Informatics FERI, University of Maribor, Smetanova ulica 17, SI-2000 Maribor, Slovenia*
saso.karakatic@uni-mb.com

***Abstract*—In this paper we research the possibility of automated combinational logic circuit (CLC) design using evolutionary computation. We propose and develop a genetic programming method which is able to construct a CLC based on the given truth tables, where the focus is to minimize the number of logic gates while accuracy is not compromised. We tested the proposed approach and compared the results both with MGA and NGA automatic methods as well as with the results obtained by human designers. Results show that our algorithm is superior to other methods as it can find correct circuits with fewer specified elements. The experiments performed on larger examples show good performance and scalability of the proposed evolutionary approach.**

***Index Terms*—Combinational circuits, design optimization, genetic programming.**

I. INTRODUCTION

One common task in digital electronics consists of designing a combinational logic circuit (CLC) that performs a desired function, given a certain specified set of available logic gates [1]. In order to fulfil the ever-growing optimization requirements such a circuit should be composed of as few elements as possible. Namely, the increase of integration level and size of integral circuits during functionality modelling of such circuits represent a problem for circuit designers. During the years many rules and techniques for finding solutions to the problem of optimizing CLC design have emerged [2]. However, the task is still demanding for a human designer, especially when a complex circuit is in question. To support the designers in performing this task different automatic methods that tackle this problem have been developed, with evolutionary computation being the most successful one [3]. While most of the research, including the leading researcher in this field Coello [4], already showed how correct CLCs can be constructed using genetic algorithms (GA), our goal was to use genetic programming (GP) method. In contrast to other research done in this field, we expanded the objective of our research to the optimization of the number of gates in the final circuits, which has not been done before. Following this path, we developed a new genetic programming method that is aimed at designing 100% correct CLCs using the specified set of available logic gates while minimizing the number of logic gates used. In this paper, we present this method and some results which show that our method is able to find solutions, that are comparable with both, human designers and other existing automatic approaches, in an efficient manner.

Manuscript received January 14, 2013; accepted March 14, 2013.

II. RELATED WORKS

The design process for CLC has evolved from its first notions to a standard element of undergraduate computing curricula [5]. Standard graphical design aids such as Karnaugh Maps [6] are widely used and tools suitable for computer implementation have evolved from the Quine–McCluskey method to freely available tools and commercial products. Soon, the researchers started to develop approaches and methods for automated design of CLC. It has turned out that the use of evolutionary techniques is one of the most viable alternatives for performing this task [7]. Literature review reveals several attempts to use evolutionary techniques for designing electrical circuits [8], some of them address optimization of digital circuits with genetics based methods [9], but only a few researchers are working on the design of circuits at the gate-level.

In some earlier researches genetic programming has been used for the design of CLCs by Louis [10], who combined GP with knowledge-based systems, and Koza [11], who focused on generating functional circuits rather than optimizing their size. Later Thompson et al. [4] focused on the configuration of a FPGA using GA, whose work influenced many other researchers working at the gate level, including Coello [4], [7], [12], who achieved good results by using GAs and multi objective design of CLC but without focusing on the number of the gates.

More recent research in this field is from Brajer [13] where she used a Cartesian GP (CGP) paradigm to construct the CLC. Instead of the standard tree structure of genotype usually used with the GP, CGP uses the array of strings for the genotype, similar to the standard GA. Her method was efficient in the construction of the CLC but again, there was no focus to optimize the number of gates used and no testing instances were provided for comparison with other methods. Same CGP approach was used by Miller [14] but again no testing instances were provided and his research was more aimed at comparing CGP to regular GA and Probabilistic Hillclimbers as a method to solve Boolean functions.

Our objective here is to automatically construct a CLC that accurately performs the desired function (specified by a given truth table) but with the strong focus on minimizing the number of gates using the standard GP approach.

III. Genetic Programming

Genetic programming is a systematic method for getting computers to automatically solve a problem starting from a high-level statement of what needs to be done [15]. GP is a domain-independent method that genetically breeds a population of computer programs to solve a problem. Specifically, GP iteratively transforms a population of computer programs into a new generation of programs by applying analogs of naturally occurring genetic operations. This process is illustrated in Fig. 1.

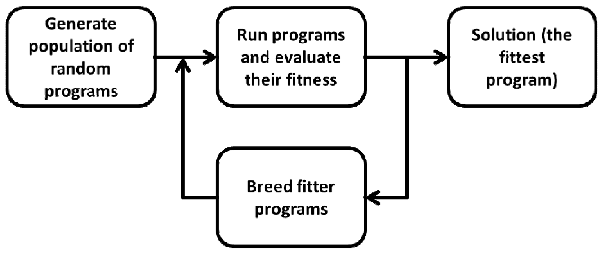
Generate population of random programs
Run programs and evaluate their fitness
Solution (the fittest program)
Breed fitter programs

Fig. 1. Main loop of genetic programming [15].

The genetic operations include crossover (recombination), mutation, reproduction, gene duplication, and gene deletion. GP is an extension of the genetic algorithm [6], in which the structures in the population are not fixed-length character strings that encode candidate solutions to a problem, but programs that, when executed, are the candidate solutions to the problem.

Programs are expressed (genotype) in GP as syntax trees rather than as lines of code. For example, the simple expression

$$((A \text{ OR } B) \text{ AND } C) \text{ XOR } (A \text{ AND } B) \tag{1}$$

is represented as shown in Fig. 2. The tree includes nodes and links. The nodes indicate the instructions to execute. The links indicate the arguments for each instruction. In this manner, the internal nodes in a tree are operators (or functions), while the tree's leaves are operands (or terminals).

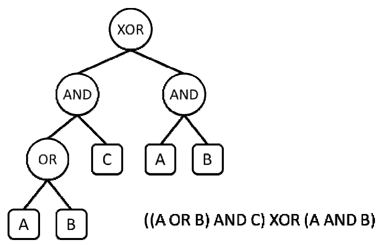

Fig. 2. An example of a syntax tree representing an expression within GP.

Some researches by Fogel [16], [17] suggest that GP outperforms GA in the sense of premature convergence, which is one of the main reasons for using GP in the present study. Fogel showed that GA may prematurely stagnate by getting a result that is not even the local optimum, whereas GP has a significantly higher chance to find the global optimum, not get caught in the local optimum and thus get better results than its counterpart evolutionary method GA. In the case of automatic CLC design, a 100% correct circuit represents a local optimum. Any further evolution that would eventually lead to the reduction of used gates normally compromises the circuit's accuracy which prevents GA from finding the global optimum. According to Fogel a properly implemented GP should outperform GA in this matter and that is why we decided to use GP approach.

IV. Implementation of the Genetic Programming System

Any CLC can be represented in the form of an expression (a formula), consisting of a set of operators (logic gates, like AND, OR, etc.) and operands (logic inputs to a circuit). The result of the expression represents the logic output of the CLC. This makes the construction of CLC a viable case to be used with GP. An expression that defines a circuit represents a program in a form of a syntax tree, which is the basic representation of a genotype in GP.

There are two conditions which each generated starting random solution must satisfy: 1) it should be unique to enforce the diversity of the population, and 2) all operands must be included in the starting solution, which eliminates the chances of too small trees in the beginning.

The basis of GP is the process where generations of solutions (syntax trees representing logic expressions) evolve towards a global objective (Fig. 1). The evolutionary process consists of evaluating the fitness of each single solution within a generation and applying of genetic operators: selection, crossover and mutation.

Standard operator used in our implementation were binary tournament selection method, automatic advancement of the elite (5% of the fittest solutions) in the new generation, crossover of two trees as shown on Fig. 3 and the mutation. Mutation operator selects a random node which undergoes one of three mutation processes based on the type of the node selected. If randomly selected node is a type of operator NOT, this node is removed from the tree. If the selected node is any other type of operator, it is either replaced by a randomly chosen different operator or deleted along with its children and a randomly selected operand comes in its place. If selected node is an operand, it is either replaced by a randomly chosen different operand or a new randomly generated sub-tree comes in its place. The mutation is depicted on Fig. 4.

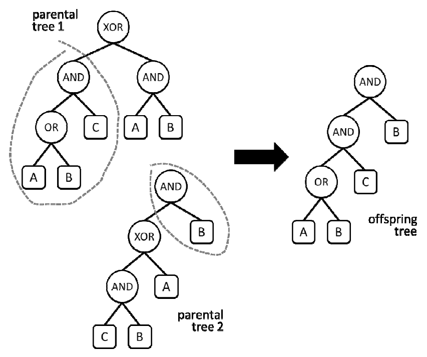

Fig. 3. An example of crossover, where offspring tree is composed out of two parts of two parental trees.

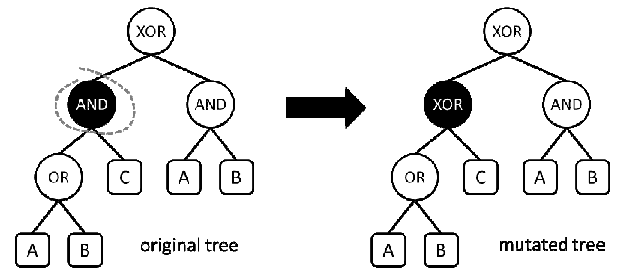

Fig. 4. An example of mutation, where a randomly chosen node of a tree is mutated) into another gate or input.

Fitness function. New offspring trees are then evaluated – their fitness is calculated. Firstly the gates are counted and secondly Boolean formula is formed from the genotype tree. Its results are compared to correct values from the truth table. The fitness value is a weighted sum of the number of incorrect results and gates

$$FF = w_r \times errors + w_g \times gates, \tag{2}$$

where *errors* is the number of rows in truth table, where the solution makes a mistake, and *gates* is the number of used gates (i.e. number of internal nodes in a syntax tree). The sum is weighed in order to give preference to correct solutions in regard to short expressions. In this manner, the algorithm searches for a solution that gives a correct output for each given combination of inputs and is also small (i.e. using the lowest possible number of logic gates).

V. EXPERIMENTS

For the purposes of this paper, three examples were chosen to illustrate our approach and the results produced are presented. Two of them are rather simple, having four inputs and one output. The resulting circuits, obtained with our proposed system, have been compared to existing solutions. The third one is a more complex one, having 6 inputs and one output; it has been used primarily for performance and scalability analysis, and also to show that the proposed system is capable to cope with more complex examples. In all three examples the allowed set of operators (gate types) was: AND, OR, XOR and NOT. Resulting circuits found by our system were compared to the MGA, the NGA, the results produced by the human designer (using Karnaugh Maps) and Sasao. Our resulting solutions are comparable in size to other solutions even when we convert all the circuits to NAND or NOR only solutions. Our GP algorithm can be adjusted to use only those two gate types, but the comparison would not be fair, as other researchers were not building circuits with only those two gates. To allow for direct comparison we used the same set of gate types as other researchers.

A. *Example #1*

Our first example has 4 inputs and one output. The comparison of the results produced by our system (Fig. 5), two other evolutionary based systems – the MGA [12] and the NGA [18], a human designer using Karnaugh Maps, and Sasao's approach [3] are shown in Table I. Sasao has used this circuit to illustrate his circuit simplification technique based on the use of ANDs & XORs. His solution uses, however, more gates than the circuit automatically produced by the NGA, MGA or our system. It can be seen that our system achieves the lowest number of used gates.

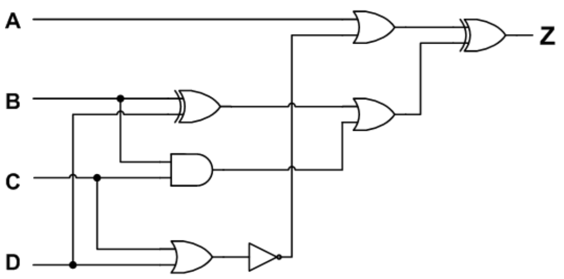

Fig. 5. The combinational logic circuit, evolved with the proposed system for the example #1.

TABLE I. BEST CIRCUITS FOR THE EXAMPLE #1.

Design	**Size**	**Resulting Circuit**
Our system	7	Z=((BC)+(D⊕B))⊕(A+(D+C)')
MGA	8	Z=(((B⊕BC)⊕((A+C+D)⊕A))'
NGA	10	Z=(BDC'⊕((B+D))⊕A⊕(C'D'A)))'
Human 1	11	Z=((A'C)⊕(D'B'))+((C'D)(A⊕B')
Human 2	12	Z=C'⊕D'B'⊕CD'A'⊕C'D'B

B. *Example #2*

Our second example is again a standard benchmark and has 4 inputs and one output. The comparison of the results produced by our system (Fig. 6), the MGA, the NGA and two human designers (the first, using Karnaugh Maps and the second using the Quine-McCluskey Procedure) are shown in Table II. It can be seen that all three genetic approaches achieve the same (lowest) number of gates.

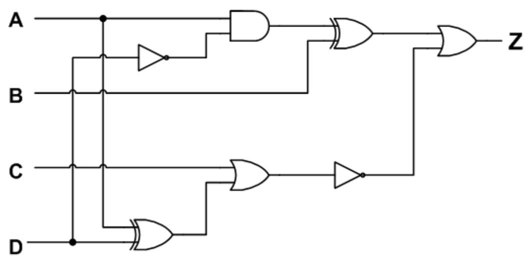

Fig. 6. The combinational logic circuit evolved with the proposed system for the example #2.

TABLE II. BEST CIRCUITS FOR THE EXAMPLE #2.

design	**size**	**resulting circuit**
our system	7	Z=(C+(D⊕A))'+(B⊕(AD'))
MGA	7	Z=((A⊕B)⊕AD)+(C+(A⊕D))'
NGA	7	Z=((B⊕A)⊕AD)+(C+(D⊕A))'
human 1	9	Z=((A⊕B)⊕((AD)(B+C)))+((A+C)+D)'
human 2	10	Z=A'B+A(B'D'+C'D)

C. *Example #3*

The third example is a more complex one, with 6 inputs and one output. We primarily generated this example for the purpose of the performance and scalability analysis. Our system was able to find a correct circuit that consisted of 46 logic gates within 300 generations, where the population size was 1.000 trees.

Let us take a look at the evolution of the accuracy (Fig. 7) and number of elements (Fig. 8) and compare them. As expected, the accuracy of the best individual rises rather fast to the point near the 150th generation, where it slows down but continues to improve, to the point where it reaches the 100% accuracy mark near 300th generation.

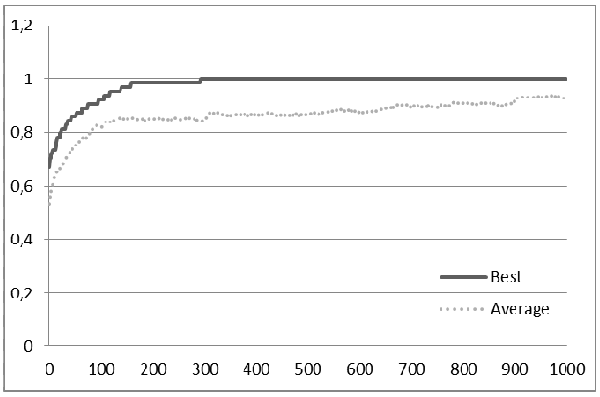

Fig. 7. Accuracy of circuits through the evolution for the example #3.

Reaching the perfect accuracy sometimes means that the number of elements must rise, as is seen in Fig. 8, where in the 292th generation the number of elements dramatically rises when the perfect accuracy is achieved. This is because fitness function prefers the accuracy over the number of elements, as mentioned earlier. For a short period, the number of elements in the best solution is even greater than the average number of elements, again due to the complex fitness function, which favors accuracy.

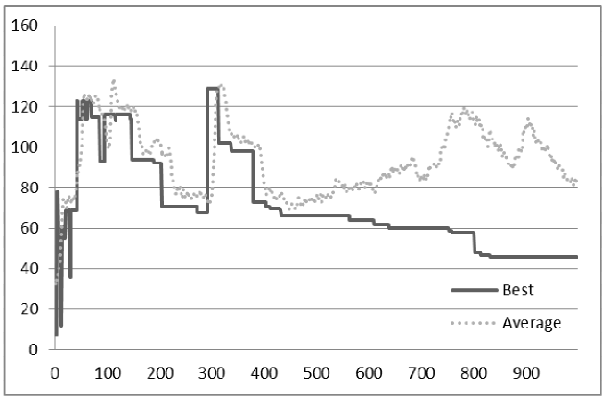

Fig. 8. Number of elements in the circuits through the evolution for the example #3.

Process does not stop here, even after the accurate solution is found, algorithm improves the size of the circuit by shortening it from 129 elements, all to the 832th generation, where it reaches the optimum with 46 elements. The test took about 5 minutes on an average desktop PC, 160 seconds in the best case and 10 minutes in the worst case scenario – it depends on the starting randomly generated generation. Because no benchmark exists, we cannot conclude if the solution is the global optimum or just a local one, but as far as our algorithm goes, this was the best solution it found.

VI. CONCLUSIONS

In this paper we presented a new GP method for automatic construction of CLCs. Our main goal was the development of a method that provides perfectly accurate circuits with the lowest possible number of specified logic gates. The obtained results show that our method was able to find such solutions – all the circuits were perfectly accurate and composed of lower or at least the same number of gates compared to the best known existing solutions. The use of GP and the representation of CLCs in a form of syntax trees within GP have enabled evolutionary search to escape local optima by maintaining the needed diversity and thus avoiding premature convergence.

At this point the genetic parameters and weights in fitness function have been determined experimentally. In the future we plan to perform exhaustive analysis of the presented method in regard to different parameter settings. Additionally, we will test our algorithm with regard to scalability and performance on large circuits.

REFERENCES

[1] R. C. Jaeger, T. N. Blalock, *Microelectronic Circuit Design*, 3rd ed. McGraw-Hill, 2000, pp. 1–32.

[2] T. Sasao, *Logic Synthesis and Optimization*. Kluwer Academic Press, 1993. [Online]. Available: http://dx.doi.org/10.1007/978-1-4615-3154-8

[3] A. Thompson, I. Harvey, P. Husbands, “Unconstrained evolution and hard consequences”, *Lecture Notes in Computer Science*, vol. 1062, pp. 136–165, 1996. [Online]. Available: http://dx.doi.org/10.1007/3-540-61093-6_7

[4] C. A. C. Coello, A. D. Christiansen, A. H. Aguirre, “Automated Design of Combinational Logic Circuits Using Genetic Algorithms”, in *Proc. of the International Conference on Artificial Neural Nets and Genetic Algorithms*, 1997, pp. 335–338.

[5] C. H. Roth, *Fundamentals of logic design*, 4th ed., West Publishing Company, 1992.

[6] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control and artificial intelligence*. MIT Press, 1992.

[7] C. A. C. Coello, A. D. Christiansen, A. H. Aguirre, “Towards automated evolutionary design of combinational circuits”, *Computers & Electrical Engineering*, vol. 27, no. 1, pp. 1–28, 2000. [Online]. Available: http://dx.doi.org/10.1016/S0045-7906(00)00004-5

[8] H. Kitano, J. A. Hendler, *Massively Parallel Artificial Intelligence*. MIT Press, 1994.

[9] M. Nirmala Devi, N. Mohankumar, S. Arumugam, “Modeling and analysis of neuro–genetic hybrid system on FPGA,” *Elektronika ir Elektrotechnika (Electronics and Electrical Engineering)*, no. 8, pp. 69–74, 2009.

[10] S. J. Louis, “Genetic algorithms as a computational tool for design”, Indiana University, 1993.

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

[12] C. A. C. Coello, L. Nacional, I. Avanzada, A. H. Aguirre, B. P. Buckles, “Evolutionary Multiobjective Design of Combinational Logic Circuits,” in *Proc. of the Second NASA/DoD Workshop on Evolvable Hardware*, 2000, pp. 161–170.

[13] I. Bajer, D. Jakobović, “Automated Design of Combinatorial Logic Circuits”, *in Proc. of the 35th International Convention MIPRO,* 2012.

[14] J. F. Miller, “An empirical study of the efficiency of learning boolean functions using- a Cartesian Genetic Programming approach”, *in Proc. of the Genetic and Evolutionary Computation Conference*, 1999, vol. 2, pp. 1135–1142.

[15] J. R. Koza, R. Poli, “Genetic Programming”, *Search Methodologies*, 2005, pp. 127–164.

[16] D. B Fogel, “Asymptotic convergence properties of genetic algorithms and evolutionary programming: analysis and experiments”, *Cybernetics and Systems: An International Journal*, vol. 25, no. 3, pp. 389-407, 1994. [Online]. Available: http://dx.doi.org/10.1080/01969729408902335

[17] D. B Fogel, “A Comparison of Evolutionary Programming and Genetic Algorithms on Selected Constrained Optimization Problems”, *Simulations*, vol. 64, no. 6, 1995.

[18] C. A. C. Coello, A. D. Christiansen, A. H. Aguirre, “Use of Evolutionary Techniques to Automate the Design of Combinational Circuits”, pp. 1–25, 1999.