Reusing Code in Genetic Programming

Edgar Galván López[1], Riccardo Poli[1], and Carlos A. Coello Coello[2]

[1] University of Essex, Colchester, CO4 3SQ, UK
egalva,rpoli@essex.ac.uk
[2] Depto. Ing. Eléctrica, Sección de Computación
Av. Instituto Politécnico Nacional No. 2508
Col. San Pedro Zacatenco, México, D.F. 07300, MEXICO
ccoello@cs.cinvestav.mx

Abstract. *In this paper we propose an approach to Genetic Programming based on code reuse and we test it in the design of combinational logic circuits at the gate-level. The circuits evolved by our algorithm are compared with circuits produced by human designers, by Particle Swarm Optimization, by an* n*-cardinality GA and by Cartesian Genetic Programming.*
Keywords: genetic programming, code reuse, logic circuit design, evolvable hardware.

1 Introduction

In Genetic Programming (GP) [7,10] programs are expressed as syntax trees. This form of GP has been applied successfully to a number of difficult problems like image enhancement, magnetic resonance data classification, etc. [2]. The reuse of code is a very important characteristic in human programming. So, several attempts have been made to introduce the ability to reuse code in GP.

For example, one can reuse code using Automatically Defined Functions (ADFs) [7,8]. The problem with this approach is discovering good ADFs. ADFs behave differently in different parts of a program when they have different arguments. So, in order to discover if an ADF is good, GP has to spend additional computation to discover with which parameters the ADF can be used properly. Code reuse is also possible with Parallel Distributed Genetic Programming (PDGP) [14,15,16]. Programs are represented in PDGP as graphs with nodes representing program primitives and links representing the flow of control and results. So, PDGP can be used to either evolve parallel programs or to produce sequential programs with shared (reused) subtrees. Another technique to reuse code is the Multiple Interacting Programs (MIPs) approach proposed in [1]. A MIPs individual is equivalent to a neural network where the computation performed in each unit is not fixed but is performed by an evolved equation. Each unit's equation is represented by a GP syntax tree.

The design of combinational logic circuits is generally considered an activity that requires certain human creativity and knowledge. Traditionally this has been performed with techniques based on Boolean algebra, for example: Karnaugh Maps [6,18], the Quine-McCluskey Algorithm [21,17] and ESPRESSO [3].

More recently the problem of designing combinational logic circuits has been attacked with various evolutionary techniques, an area called *Evolvable Hardware* [7,19,20]. Evolvable Hardware research can be sub-divided into two main categories: intrinsic evolution, which is carried out through building and testing electronic hardware, and extrinsic evolution, carried out through simulations. Extrinsic evolution is the approach used in the work presented here.

Louis [12] was one of the first to use genetic algorithms for combinational logic design. In his thesis [11], Louis uses a genetic operator called *masked crossover* which adapts itself to the encoding and is able to exploit specific information about the problem domain.

Coello *et al.* [4] developed an approach using a two-dimensional matrix in which each matrix element is a gate (AND, OR, NOT, XOR, WIRE). The goal was to produce fully functional designs which were also minimal. They tried to achieve this by maximizing the use of WIRE gates, once feasible circuits have been found. In this work the authors reported good result for small problems. This approach, however, is highly sensitive to the values of the parameters adopted, namely the size of the matrix which, if not chosen properly, may prevent the GA from converging.

Coello *et al.* [5] used the same circuit representation as in the work mentioned above but this time used Particle Swarm Optimization (PSO) to design the circuits. Instead of using the usual PSO's real-valued representation, they used a binary encoding in order to facilitate representing circuits. The algorithm produced competitive results with respect to an n-cardinality Genetic Algorithm (NGA), but its performance seriously degraded when dealing with circuits with more than one output.

When appropriate terminals, functions and fitness functions are defined, standard GP can go beyond the production of programs and can be used to evolve circuits. The approach in [9] was, for example, to use primitives that can grow a circuit starting from an initial embryonic circuit including only a power supply, a load, and a modifiable wire. This approach has been mostly used to evolve analogue circuits and controllers.

Miller *et al.* [13] developed a technique called Cartesian Genetic Programming (CGP). In CGP a program is seen as a rectangular array of nodes. Each node may implement any convenient programming construct. In this representation, all cells are assumed to have three inputs and one output and all cell connections are feed-forward. With this technique, Miller *et al.* obtained good results for complex problems, including the three-bit multiplier and the even-*4* parity circuit. However, the approach normally requires a very high number of fitness function evaluations.

The main purpose of the present work is to explore a new GP technique, called *Encapsulated GP* (EGP), which allows the reuse of code within GP trees. In this work we will use evolvable hardware problems to test EGP. The paper is organized as follows. Section 2 contains a description of the algorithm used. We provide experimental results in Section 3. We discuss these in Section 4 and we draw some conclusions in Section 5.

2 Encapsulation Genetic Programming

2.1 Terminal and Functional Set

The representation used in our work is a tree-like one as suggested by Koza [7]. The function set for circuits with more than one output is {*AND, OR, NOT, XOR*}. For practical reasons, the functions are internally represented with numbers as indicated in Table 1. Boolean algebra notation (see third column in Table 1) is instead used to compare the results found by EGP with those produced with others techniques. In this notation the absence of a symbol between two terminals indicates the presence of an *AND*. Also, we use the relations A *NOR* B = *NOT*(A *OR* B) and A *NAND* B = *NOT*(A *AND* B) to represent NOR and NAND, respectively.

The terminal set consists of the inputs of the circuit. These are defined with the letters {*a,b,c,...*}. The terminal set includes also a special *encapsulation terminal*, *p*, which will be explained later. We use the *"Grow"* initialisation method, which allows the selection of nodes from the whole primitive set until the depth limit is reached.

Once we have defined the terminal and functional sets, we proceed to generate the individuals in the population. For this we have used the *postfix* representation. We chose this representation because it is easy for the computer to execute, using a stack-based interpreter (see Section 2.2).

2.2 Interpreter of Genomes

To evaluate each individual, it is necessary to use an expressions' evaluator (interpreter of genomes). Ours is based on a stack, and works in the following way. It reads characters from left to right. If the character read is a terminal, then its corresponding value is stored in a stack. If the character read is a function (see Table 1), then the values for its arguments will be taken from the stack for evaluation and the result of this evaluation will be stored on the top of the stack (see Figure 1). This procedure is repeated until the end of the expression is reached.

Table 1. Function set (first column). The second column reports the character used in our implementation, while the third shows the corresponding Boolean algebra notation.

Boolean Operators	*Representation*	*Symbol*
NOT	1	'
OR	2	+
NOR	3	(see text)
AND	4	(see text)
NAND	5	(see text)
XOR	6	$\oplus$

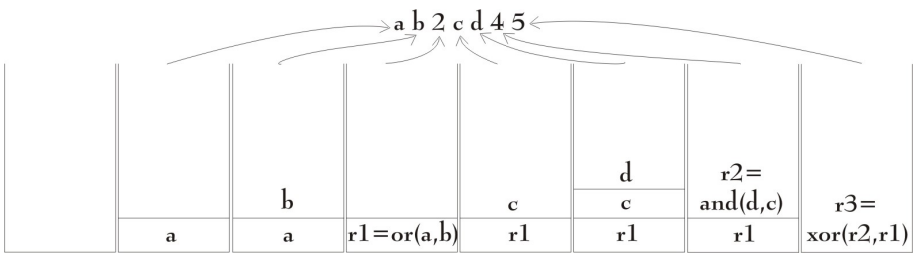

Fig. 1. Contents of the stack at different stages of the interpretation of the program `ab2cd45`.

2.3 Encapsulation Terminal

The method proposed in this paper does not only allow reusing code but it also allows evolving graph-like structures, which could encode, for example, combinational logic circuits. This is the result of using the p terminal symbol, which works as follows:

- Once the individuals in the population have been generated, every individual is checked to see if it contains p's at the genotype level. If an individual contains this symbol, we assign one point within the individual to which this p refers.
- If the p symbol points to a function symbol, the p symbol effectively represents the sub-tree rooted at that function.
- If the p symbol points to a terminal symbol, the p symbol simply represents that node.

2.4 Genetic Operators

The genetic operators used in EGP are: tournament selection, crossover, mutation and elitism.

The crossover operator works as usual: 1) two individuals are selected from the population; 2) we randomly select a crossover point in each parent; 3) we swap the sub-trees rooted at the crossover points. An important difference is that, if the sub-tree swapped contained a p symbol, the p symbol's pointer is *not* changed (Figure 2 illustrates this behaviour).[3] This means that, as a result of crossing over, the code represented by p's *may* be in individuals different from the one containing the p symbols. Figure 3 shows the typical pattern of reuse resulting from repeated applications of crossover.

The mutation operator works as follows: 1) an individual is selected in the population; 2) a random node is selected; 3) the node is replaced with a different random primitive, taken from the primitive set. In our algorithm, if the node

[3] There is an exception to this rule: we prevent a p symbol from referring to a sub-tree that contains the same p since this would lead to an infinite loop. We do this by reassigning the positions to which the p in question is pointing to.

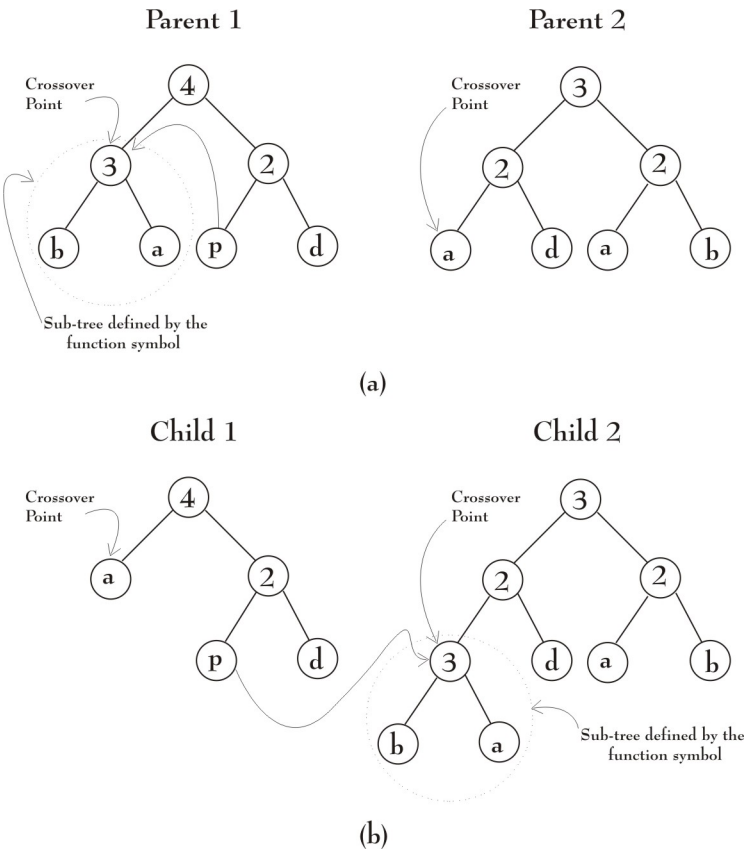

Fig. 2. Two parent trees *before* applying the crossover operator (a) and the children obtained *after* applying the crossover operator (b).

selected contains a reference point from a p symbol, then we replace first this p with the sub-tree it points to and then, we apply the mutation operator.

Elitism is in charge of guaranteeing that the best individual in the current generation passes intact to the following generation.

2.5 Fitness Function

The fitness function that we used for circuit design works in two stages: at the beginning of the search, the fitness of a genotype is the number of correct output bits (raw fitness). Once the fitness has reached the maximum number of correct outputs bits, we try to optimize the circuits by giving a higher fitness to individuals with shorter encodings.

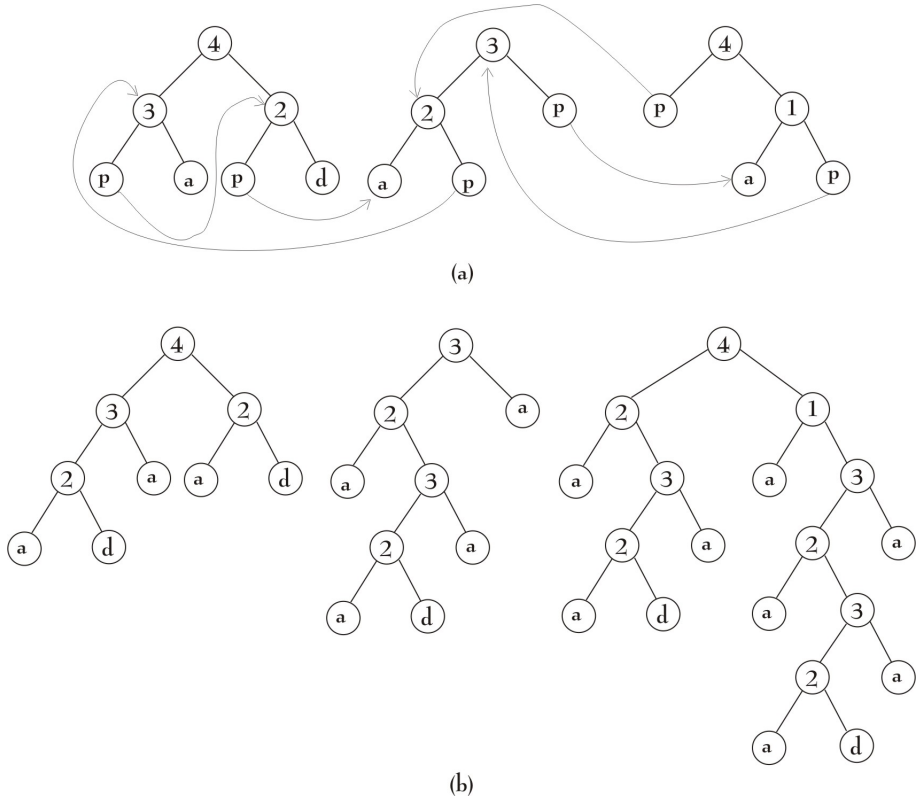

Fig. 3. Example of typical pattern of code reuse in EGP. After a few generations individuals contain numerous p symbols pointing at each other's code (a). The corresponding, logically-equivalent population shows a high degree of similarity between individuals (b).

3 Experimental results

We used several evolvable hardware problems of different complexity taken from the literature to test EGP. Our results were compared with those obtained by human designers, by NGA [4], by PSO [5] and by CGP [13].

After a series of preliminary experiments we decided to use a crossover rate of 70% and a mutation rate of 30%. In all runs we kept the best individual of each generation (elitism). For all the examples, the we performed 20 independent runs.

3.1 Two-bit adder

Our first example is a two-bit adder with 4 inputs and 3 outputs.

The parameters used in this example are the following: population size = 560, maximum number of generations = 700.

Table 2. Comparison of results between an NGA, Karnaugh Maps plus Boolean algebra, the PSO algorithm and EGP on the two-bit adder problem.

NGA	**PSO**
$F_1 = B \oplus D$	$F_1 = B \oplus D$
$F_2 = (A \oplus C) \oplus BD$	$F_2 = (BD) \oplus (A \oplus C)$
$F_3 = AC + BD(A \oplus C)$	$F_3 = (AC) + ((BD)(A \oplus C))$
7 gates	7 gates
2 ANDs, 1 OR, 4 XORs	3 XORs, 3 ANDs, 1 OR
Karnaugh Maps plus Boolean algebra	**EGP**
$F_1 = A \oplus D$	$F_1 = AC + ((BC)(A \oplus C))$
$F_2 = (A \oplus C)D' + ((A \oplus C) \oplus B)D$	$F_2 = (BC) \oplus (A \oplus C)$
$F_3 = AC + BD(A + C)$	$F_3 = B \oplus D$
12 gates	7 gates
5 ANDs, 3 ORs, 3 XORs, 1 NOT	3 ANDs, 3 XORs, 1 OR

All the runs found the feasible region[4]. That is in all runs the program could solve the problem. The best result found in this example contained 7 gates (3 ANDs, 3 XORs, 1 OR), and 10% of the runs found circuits with this number of gates.

The results produced by EGP in this example were competitive w.r.t. those produced using Karnaugh Maps, *NGA* and PSO (see Table 2).

3.2 Two-bit multiplier

Our second example is a two-bit multiplier with 4 inputs and 4 outputs.

The parameters used in this example are the following: population size = 450, maximum number of generations = 500.

Again, we obtained interesting results. All the runs performed were able to reach the feasible region. The best result found in this example contained 7 gates (5 ANDs, 2 XORs) and 15% of the runs found circuits with this number of gates.

The results produced by EGP in this example were compared against those produced using Karnaugh Maps plus Boolean algebra, the NGA and CGP (see Table 3).

3.3 Katz

Our third example is the Katz circuit with 4 inputs and 3 outputs.

The parameters used in this example are the following: population size = 880, maximum number of generations = 4,000.

In 30% of the runs EGP found the feasible zone. The best result found in this example contained 19 gates (4 AND, 7 XORs, 4 ORs, 4 NOTs) and 5% of the runs found circuits with this number of gates.

[4] The feasible region is the area of the search space containing circuits that match all the outputs of the problem's truth table.

Table 3. Comparison of results between the NGA, Karnaugh Maps plus Boolean algebra, CGP and EGP on the two-bit multiplier problem.

NGA	**CGP**
$F_1 = (BD)(AC)$	$F_1 = (AD)(BC)$
$F_2 = AC \oplus ((BD)(AC))$	$F_2 = ((AD)(BC)) \oplus AB$
$F_3 = BC \oplus AD$	$F_3 = AD \oplus BC$
$F_4 = BD$	$F_4 = CD$
7 gates	7 gates
5 ANDs, 2 XORs	5 ANDs , 2 XORs
Karnaugh Maps plus Boolean algebra	**EGP**
$F_1 = (AB)(CD)$	$F_1 = (AD)(BC)$
$F_2 = AC(BD)'$	$F_2 = ((AD)(BC) \oplus A)C$
$F_3 = BC \oplus AD$	$F_3 = AD \oplus BC$
$F_4 = BD$	$F_4 = BD$
8 gates	7 gates
6 ANDs, 1 XOR, 1 NOT	5 ANDs, 2 XORs

The results produced by our system in this example were compared against those produced using Karnaugh Maps plus Boolean algebra and Quine-McCluskey Procedure (see Table 4).

4 Discussion

On the test problems reported above, our algorithm has shown very competitive results with respect to other approaches that employ other types of evolutionary algorithms. Although results are not improved with respect to those previously reported, our approach consistently reaches the feasible region (which does not always happens with other methods). We believe the good performance of EGP is largely due to its ability to reuse code (as indicated by the large number of p's used in all the solutions found). In tests with larger (and more difficult) circuits, the algorithm has shown promise, but more tests are needed in order to allow a fair assessment of its performance.

5 Conclusions and Future Work

We have presented EGP, a new genetic programming approach to evolve programs with a high degree of code reuse. The approach has been validated using test functions taken from the evolvable hardware literature. Comparison between EGP and other heuristics has shown that our approach can consistently reach the feasible region of the three test problems used while converging to the best-known solutions for two of them.

In our future work we plan to extend our algorithm in several ways. The ideas behind our algorithm are general and, thus, EGP can be adopted in other

Table 4. Comparison of results between Karnaugh Maps plus Boolean algebra, Quine-McCluskey Procedure and EGP on the Katz problem.

Karnaugh Maps plus Boolean algebra
$F_1 = (A\oplus)'(B \oplus D)'$
$F_2 = B'D(A' + C) + A'C$
$F_3 = BD'(A + C') + AC'$
19 gates
2 XORs, 4 ORs, 7 ANDs, 6 NOTs
Quine-McCluskey Procedure
$F_1 = (A \oplus C)'(B \oplus D)'$
$F_2 = A'C + (A \oplus C)'(B'D)$
$F_3 = (F_1 + F_2)'$
13 gates
2 XORs, 2 ORs, 4 ANDs, 5 NOTs
EGP
$F_1 = (A \oplus C')(B \oplus D')$
$F_2 = (C + (D' + D)')(((C \oplus D) + B)A)'$
$F_3 = (((DB \oplus A) \oplus B) + (C \oplus A)) \oplus C$
19 gates
4 ANDs, 7 XORs, 4 ORs, 4 NOTs

application domains in which code reuse (and/or graph-like representations) may be beneficial. Also, we are interested in extending EGP by adding self-adaptation mechanisms that would make preliminary runs and parameter tuning unnecessary. We would also like to in incorporate multi-objective optimization concepts to improve the search capabilities of EGP.

Acknowledgements

The first author thanks the Mexican Consejo Nacional de Ciencia y Tecnología (CONACyT) for support to pursue graduate studies at University of Essex. The third author also acknowledges support from CONACyT (project No. 34201-A). Finally, the Essex authors would like to thank the members of the NEC (Natural and Evolutionary Computation) group for helpful comments and discussion.

References

1. P. J. Angeline. Multiple Interacting Programs: A Representation for Evolving Complex Behaviors. *Cybernetics and Systems*, 29(8):779–806, November. 1998.
2. W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming An Introduction*. Morgan Kaufmann Publishiers, Inc, San Francisco, CA, 1998.
3. R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

4. C. A. C. Coello, A. D. Christiansen, and A. H. Aguirre. Use of Evolutionary Techniques to Automate the Design of Combinational Circuits. *International Journal of Smart Engineering System Design*, 2(4):229–314, June. 2000.
5. C. A. C. Coello, E. H. Luna, and A. H. Aguirre. Use of particle swarm optimization to design combinational logic circuits. In *Evolvable Systems: From Biology to Hardware. 5th International Conference, ICES 2003*, volume 2606, pages 398–409, Trondheim, Norway, March. 2003. Springer, Lecture Notes in Computer Science.
6. M. Karnaugh. A map method for synthesis of combinational logic circuits. *Transactions of the AIEE, Communications and Electronics*, 72(I):593–599, November. 1953.
7. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, Massachusetts, 1992.
8. J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. The MIT Press, Cambridge, Massachusetts, 1994.
9. J. R. Koza, F. H. Bennet, D. Andre, and M. A. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kauffman Publishiers, Inc, 1999.
10. W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer, 2002.
11. S. J. Louis. *Genetic Algorithms as a Computational Tool for Design*. PhD thesis, Department of Computer Science, Indiana University, 1993.
12. S. J. Louis and G. J. Rawlins. Designer Genetic Algorithms: Genetic Algorithms in Structure Design. In *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 53–60, San Diego California, . 1991. Morgan Kaufmann Publishiers, Inc.
13. J. F. Miller, D. Job, and V. K. Vassilev. Principles in the Evolutionary Design of Digital Circuits - Part I. *Journal of Genetic Programming and Evolvable Machines*, 1(1):8–35, . 2000.
14. R. Poli. Some Steps Towards a Form of Parallel Distributed Genetic Programming. In *Proceedings of the 1st Online Workshop on Soft Computing*, pages 290–295, Nagoya, August. 1996.
15. R. Poli. Discovery of Symbolic, Neuro-Symbolic and Neural Networks with Parallel Distributed Genetic Programming. In *Procedures of 3rd. International Conference on Artificial Neural Networks and Genetic Algorithms, ICANNGA 1997*, pages 419–423, Norwich, April. 1997. Springer.
16. R. Poli. Parallel Distributed Genetic Programming. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 403–431, London, 1999. McGraw-Hill.
17. W. V. Quine. A way to simplify truth functions. *American Mathematical Monthly*, 62(9):627–631, November. 1955.
18. C. E. Shannon. Minimization of boolean functions. *Bell Systems Technical Journal*, 35(5):1417–1444., 1956.
19. J. Torresen. A Divide-and-Conquer Approach to Evolvable Hardware. In *2nd International Conference, ICES 1998*, volume 1478, pages 57–65, Lausanne, Switzerland, September. 1998. Springer, Lecture Notes in Computer Science.
20. J. Torresen. Evolvable Hardware - The Coming Hardware Design Method? In *Neuro-fuzzy techniques for Intelligent Information Systems*, pages 435–449. N. Kazabov and R. Kozma (editors), Physica-Verlag (Springer-Verlag), . 1999.
21. E. W. Veitch. A Chart Method for Simplifying Boolean Functions. *Proceedings of the ACM*, pages 127–133, May. 1952.