Using Genetic Programming and Multiplexers for the Synthesis of Logic Circuits

Arturo Hernández Aguirre
Department of Computer Science
Center for Research in Mathematics
Guanajuato, Gto, 36240, México
artha@cimat.mx
and
Carlos A. Coello Coello
CINVESTAV-IPN Computer Science Section
México, D.F. 07300, México
ccoello@cs.cinvestav.mx

Abstract

This paper introduces the circuit design problem as a synthesis procedure. An evolutionary technique denominated Genetic Programming is proposed as the main engine for the synthesis of logic circuits. The paper argues that the synthesis of circuits using bottom-up procedures (such as Genetic Programming) is at least as powerful as any top-down method, and that this is possible by means of the replication of a single element: the binary multiplexer. The properties of this device are described as a sound basis for the synthesis of logic circuits using Genetic Programming. Several circuits are synthesized and contrasted against two design methods: the standard implementation of logic functions using multiplexers, and ordered binary decision diagrams.

Keywords: logic synthesis, computer-aided design, optimization, evolvable hardware, genetic programming, artificial intelligence.

1 INTRODUCTION

Automated design in any domain is a task considered quite difficult to achieve with a computer, particularly if involves human decisions that do not seem to follow a well-defined pattern. The synthesis of logic circuits is one of these design problems that is difficult to automate. In previous work we have developed an approach based on a genetic algorithm (GA) to optimize combinational circuits at the gate level [9, 10]. In the aforementioned work, the design metric adopted was the

number of gates of a circuit, so that our goal was to produce fully functional circuits that required the smallest possible number of gates (chosen from a certain set defined by the user).

However, the use of this metric may not be realistic in VLSI systems design where the emphasis is to decrease the whole manufacturing cost rather than reducing the total number of components used [11]. It is common, therefore, to replicate the same unit as many times as possible, despite the fact that this may lead to circuits with a larger number of gates. Furthermore, in this domain, the silicon surface needed to implement any logical component is another very important factor that deserves consideration, and our previous work had not considered either of these issues. Additionally, we were aware of the strong bias imposed by the matrix representation that we have used in the past to design combinational circuits [8, 10] and wanted to address this issue as well. For this article, the circuit design problem is restated in such a way that the issues previously mentioned are taken into account.

This paper emphasizes the importance of replication by allowing the use of only one device: the multiplexer[1] with 1-control line. Then, we restate the problem so that the goal is the generation of fully functional circuits in which the total number of multiplexers used is minimum. The representational bias issue mentioned before is addressed by adopting the tree encoding technique used in genetic programming [2]. The underlying hypothesis of the methodology introduced by the paper is that the replication of simple and elementary binary multiplexers is a sound process for the synthesis of logic functions.

The organization of this paper is the following: Section 2 describes some previous related work. Section 3 states the problem of interest in a more detailed form. Sections 4, 5, 6, and 7 are devoted to the introduction of a methodology based on genetic programming to synthesize logic functions using multiplexers. Sections 8 and 9 present a variety of examples taken from the technical literature. To validate the approach circuits are compared in two ways: one is the comparison of the number of muxes found in the minimized circuits against the number of muxes needed by the standard implementation. The other comparison is against a popular logic synthesis approach called Ordered Binary Decision Diagrams (OBDDs). Section 10 is devoted to test the consistency of the method. Section 11 provides conclusions.

2 PREVIOUS WORK

A general search technique inspired by natural evolution, called the *genetic algorithm* (GA) [19], has been widely used for optimization tasks [17] and is known to be a very powerful tool in certain domains. Genetic Programming (GP) [23] is an extension of the GA in which a tree-based representation is used instead of the traditional linear chromosomic binary representation employed by the GA.

Probably the earliest attempt to evolve circuits is Friedman's thesis, that dates back to the mid 1950s [16]. In his thesis, Friedman proposed that a series of control circuits, similar to what we now call neural networks, could be evolved through "selective feedback" in a process analogous to

[1] Throughout this paper, we will often refer to a 1-control line multiplexer using the abbreviation "mux" (plural is "muxes").

natural selection. J. W. Atmar [5] was another early researcher to incorporate directly the bit string representing the configuration of a programmable circuit within the genotype of an evolutionary-based technique.

In the contemporary literature, the attempt to use evolutionary-based techniques to design electrical circuits has been called "evolvable hardware" [22, 12]. Within evolvable hardware there are only a few researchers working on the design of circuits at the gate-level.

Louis [28] is one of earliest sources that report the use of GAs to design combinational logic circuits. In his dissertation [27] Louis combines knowledge-based systems with the genetic algorithm, making use of a genetic operator called *masked crossover* that adapts to the encoding, being able to exploit information unused by classical crossover operators. His idea of incorporating knowledge about the domain in the genetic operator constitutes a big step toward increasing the power of the GA as a design tool. Unfortunately, the incorporation of knowledge into the GA decreases its usefulness as a *general* search tool. Louis overcomes this problem by defining an operator that he claims to be domain independent, but whose efficiency turns out to depend on the representation used.

Koza [23] has used genetic programming to design combinational circuits. He has designed, for example, a two-bit adder, using a small set of gates (AND, OR, NOT), but his emphasis has been on generating functional circuits rather than on optimizing them. In fact, this is also the case in Louis' research, where the main focus was to provide an easier way to generate functional designs using the GA rather than in optimizing a functional design according to certain metrics. In more recent work, Koza [25, 24] has focused more towards the design of analog circuits in which the goal is to produce their appropriate topology and size so that they are functional given a certain set of components. So far, genetic programming has been considered a more powerful tool in such tasks, because the representation it uses is more powerful for structural design in general. However, genetic programming produces circuits that are highly redundant and difficult to simplify automatically. Furthermore, the computer resources normally required to produce such circuits are very demanding in terms of memory and CPU time.

Another early effort to codify the basic logic gates (AND, OR, and NOT) along with their possible interconnections was offered by Thompson et al. [35]. Thompson's work focuses on the configuration of a Field Programmable Gate Array (FPGA) using genetic algorithms, and was the basis for the work performed later by most of the other researchers interested in evolvable hardware at the gate level.

Miller et al. [29] developed an approach that uses a very compact representation that instead of considering the inputs and gates of a circuit as completely separate elements in the chromosomic string, uses a single gene to encode a complete Boolean expression. Miller's notation decreases the total length of the chromosome, but it increases the cardinality of the alphabet needed, having as its main drawback the lack of flexibility of the representation to handle a larger number of inputs (the cardinality of the alphabet in Miller's case grows exponentially with respect to the number of inputs).

The only other work on evolvable hardware at a gate level is the one reported by Iba et al. [21], who used a variable-length GA with an array representation. However, the emphasis of Iba's work is learning rather than optimization.

None of the previously mentioned approaches has concentrated on the exclusive use of multiplexers to design combinational circuits using evolutionary techniques, although some researchers such as Miller [29] have used multiplexers as another permissible gate which can be combined with the traditional Boolean functions to design circuits.

Several strategies for the design of combinational circuits using multiplexers have been reported after the concept of *universal logic modules* [38]. Chart techniques [26], graphical methods for up to 6 variables [36], and other algorithms more suitable for programming have been proposed [30, 18, 4, 34]. The aim of these approaches (muxes in cascade or tree, or a combination of both), is either to minimize the number of multiplexers, or to find p control variables such that a Boolean function is realizable by a multiplexer with $p-$control signals.

Our goal is a general methodology for the synthesis of any logic function specified through a truth table. No specific domain knowledge is used to modify the genetic operators, neither it needs to be known by the system.

Binary decision diagrams have been also very popular in the circuit optimization literature. Akers [3] proposed binary decision diagrams as the vehicle to represent and minimize Boolean functions. Bryant [6] proposed directed acyclic graphs for the same end. Both approaches are based on the manipulation of the nodes of the graph, thus, the initial graph is transformed into functional equivalent subgraphs while preserving the Boolean function encoded. The repetitive application of several node rules derived from the problem domain (remove terminals, remove nonterminals, remove redundant tests [7], and also *reduce*) [31] have proven sufficient to simplify binary decision diagrams, and to reduce them into ordered binary decision diagrams. In essence, the goal is achieved by a top-down minimization strategy, that is, reduced graphs are produced from complete graphs. The approach, although useful to test functional equivalence (generation of the same truth table) and other circuit properties, it does not fully minimize a circuit [20].

The evolutionary computation approach we are to describe follows essentially the opposite direction. A bottom-up searching procedure (genetic programming [23]) constructs Boolean functions by combining samples taken from the space of partial solutions. Once a 100% functional solution is found, our goal is to minimize such a solution. Thus, the fitness function is updated to reward fully functional solutions with fewer elements. Trees are therefore trimmed, and nodes are replicated without adding any heuristic other than a simple change in the fitness function.

The difference in the two approaches should be evident at this point: in graph techniques the "minimization rules" are derived from the problem domain. In our evolutionary system we work with the *purest* form of genetic programming. Thus, no problem domain knowledge was included in the evolutionary process, and yet, our approach is able to find excellent results as we will see in a further section.

Ordered Binary Decision Diagrams (OBDDs) have also been used in combination with evolutionary computation techniques. Yanagiya [37] is credited as being the first to use OBDDs to learn a 20-multiplexer with genetic programming. After him, several other researchers have performed similar research (see for example [14]). However, the emphasis of that work has been the modification of the genetic operators as to implement the standard OBDD simplification rules. They also considered the use of special techniques to create rather small initial populations that functionally agree with the truth table. Thus, the low bias in the initial population reduces the search space,

and in consequence the time to find a solution. The most obvious disadvantage of these constraints specifically designed to learn a certain Boolean function (e.g., a multiplexer) is their lack of generality. Other researchers, such as Drechsler [13] have focused on improving the learning rate of a genetic algorithm by training it with a set of examples. Although the training process turns out to be relatively slow, the performance of the genetic algorithm is considerably accelerated after deriving certain heuristics from the learning stage, and applying them to a different Boolean function. The focus of these papers is different from ours, since we do not concentrate on the problem of learning a certain Boolean function (like Droste [14]) or a certain node ordering for an OBDD (like Drechsler [13]), but in designing and optimizing combinational circuits defined by a truth table using only multiplexers [2],[1]. Nevertheless, we compare our results against those produced by OBDDs in a further section.

3 STATEMENT OF THE PROBLEM

The problem of interest to us consists of designing a circuit that performs a desired logic function (specified by a truth table), using the least possible number of 1-control line multiplexers. As will be described below, a logic function with n variables can be implemented using $2^n - 1$ muxes (called the *standard implementation* in this paper). Any implementation using less than that number of elements could be considered an improvement in the design. Since the optimal minimum number needed is unknown for most of the logic functions, the use of a heuristic such as genetic programming [23] seems adequate.

The approach of this paper allows only "1s" and "0s" to be fed into the muxes. This makes a clear difference with respect to the well-known tabular techniques used for the synthesis of logic circuits, because in those cases a variable can be fed into a multiplexer. In our case we allow the variables to be used only as the control signal of the multiplexer.

The implementation cost measured in terms of silicon surface has been studied for many years. Consider an n-variable multiplexer realized by means of $2^n - 1$ multiplexers with a 1-control signal. Assuming that the cost of a single unit is K, the cost of such a realization is proportional to $K(2^n - 1)$. Therefore, any realization with a fewer number of elements implies an improvement of the total manufacturing cost [11]. Recently, Scholl introduced pass transistor logic for the implementation on silicon of this kind of synthesized circuits [32].

4 MULTIPLEXERS AS UNIVERSAL LOGIC BASIS ELEMENTS

A *multiplexer* with n *selection lines* is a combinational circuit that selects data from 2^n input lines and directs it to a single output line. A procedure based on the residue of a Boolean function combined with Shannon's decomposition produces a circuit that only uses binary multiplexers (the complement of a Boolean variable x will be denoted x' hereafter)

Definition 1 *Residue of a Boolean Function The residue of a Boolean function $f(x_1, x_2, \ldots, x_n)$ with respect to a variable x_j is the value of the function for an specific value of x_j. It is denoted by f_{x_j}, for $x_j = 1$ and by $f_{\bar{x}_j}$ for $x_j = 0$.*

The Shannon expansion [33] in terms of residues of the function is,

$$f = \bar{x}_j f|_{\bar{x}_j} + x_j f|_{x_j} \tag{1}$$

For mapping Boolean expansions into circuits using binary multiplexers, each variable x_j in Equation 1 takes the control line of the mux. For the sake of an example consider the function $f(a, b, c) = a'b'c + a'bc' + ab'c'$.

The residue of the expansion over the variable a is:

$$\begin{aligned} f(a, b, c) &= a'f|_{a=0} + af|_{a=1} \\ &= a' \cdot (b'c + bc') + a \cdot (b'c') \end{aligned}$$

The factor $b'c + bc'$ must be taken by the mux when the selector a is "low", and $b'c'$ when a is "high". These factors could also be expanded in the same way. The expansion of the first factor over the variable b is:

$$\begin{aligned} b'c + bc' &= b'(b'c + bc')|_{b=0} + b(b'c + bc')|_{b=1} \\ &= b' \cdot c + b \cdot c' \end{aligned}$$

And the expansion of the second factor over b is:

$$\begin{aligned} b'c' &= b'(b'c')|_{b=0} + b(b'c')|_{b=1} \\ &= b' \cdot c' + b \cdot 0 \end{aligned}$$

The circuit implementing the Boolean function of our example is shown in Figure 1. Our system captures the essence of this expansion, therefore, it will produce circuits whose only inputs are 0s and 1s. Different variable order during the expansion can produce smaller circuits if some Boolean terms are shared by two or more branches of the tree. This is a combinatorial problem suitable for an evolutionary computation technique.

The *standard implementation* of a Boolean function using binary muxes is shown in Figure 2. This is, every multiplexer with n-control lines can be synthesized by $2^n - 1$ muxes with 1-control line. Notice that the number of layers or depth of the array is equal to n.

Multiplexers can be "active low" or "active high" devices, a property that we simply denote as *class A* and *class B* (this is similar to compute the Shannon expansion over a variable a, or its complement a'). For a *class A* multiplexer, when the control is set to one the input labeled as "1" is copied to the output, and viceversa, the input labeled as "0" is copied to the output when the control is zero. For a *class B* multiplexer the logic is exactly the opposite: copy the input labeled

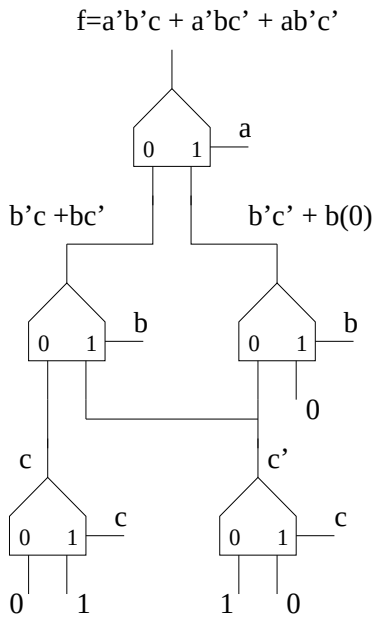

Figure 1: Shannon expansion implemented with binary multiplexers

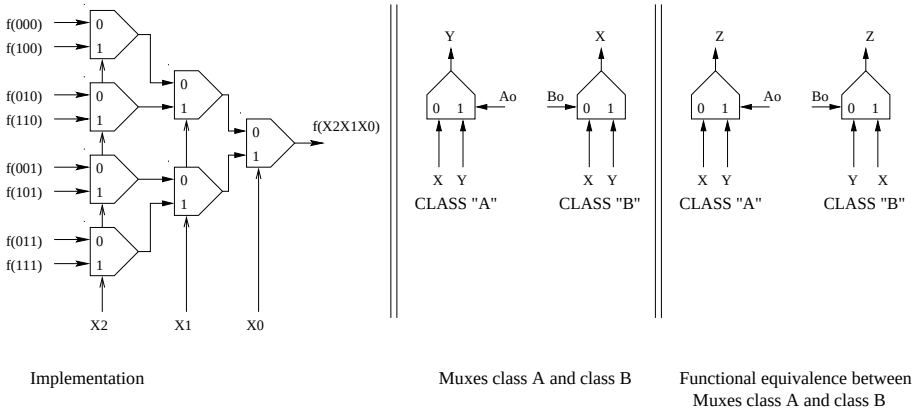

Figure 2: Implementation of a multiplexer of 3-control signals by means of 7 1-control signal muxes. Muxes class "A" and class "B". Functional equivalence between both classes

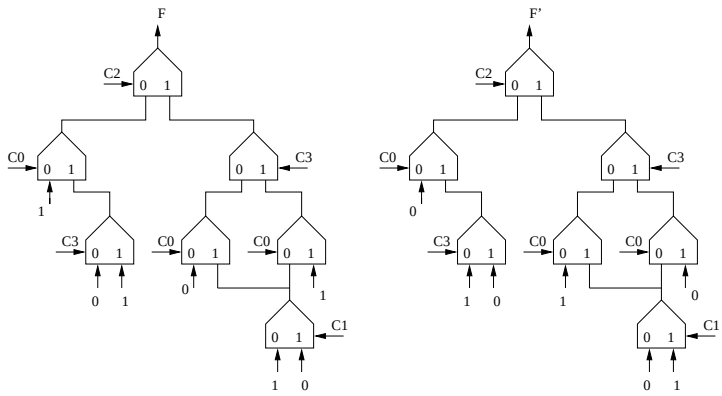

Figure 3: Transformation of a function into its complement function

"0" when the control line is one, and copy the input labeled "1" when the control is zero. In order to differentiate this property, class A muxes have the control signal on the right hand side and class B on the left, as can be seen in Figure 2. Therefore the control signal is located on the side of the input to be propagated when the control is in an *active state* (the active state will be "1" for all our examples).

It is possible to use both classes of multiplexers simultaneously in a circuit. The design criteria could allow them as well. Two characteristic properties of circuits of this nature should be taken into consideration during the design process:

- **Class Transformation Property**: Class A and class B multiplexers can be converted freely from one class into the other, by just switching their inputs, thus input labeled "1" goes to input "0" and input labeled "0" now goes into "1" (see Figure 2).
- **Complement Function Property**: For every logic function F, its complement F' is derivable from the very same circuit that implements F by just negating the inputs, that is, by changing "0s" to "1s" and "1s" to "0s" (see Figure 3) [2].

The corresponding consequences of these properties are the following:

- **Implementation in One Class**: Every circuit can be implemented by means of multiplexers of only one class (by the class transformation property). Since we are aiming to replicate the same element as many times as possible, this is a highly beneficial design quality, as can be appreciated in Figure 4 [2].
- **The Minimum Circuit Equivalence**: If the function F and its complement F' are found to be implemented by particular and different size (i.e., number of elements) circuits, then both circuits are solutions for both functions (by the complement function property). Therefore, the smallest circuit is the desirable solution. This means that in practice the designer would have an alternate procedure to verify the quality of the solution [2].

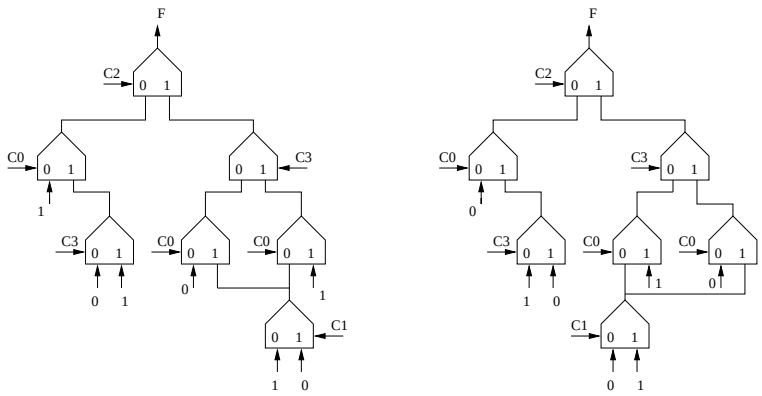

Figure 4: Transformation of a design into a circuit using multiplexers of only one class (*Class B* is shown in this example)

5 NOTIONS OF GENETIC PROGRAMMING

One of the early goals of Artificial Intelligence (AI) relates to the automatic generation of computer programs to perform a certain task. For several years, however, this goal seemed too ambitious since there is normally an exponential growth of the search space as we extend the domain of a certain program, and consequently, any technique will produce either invalid or very inefficient programs.

Some evolutionary computing techniques attempted to deal with automatic programming since their conception, but their notorious failures even in very simple domains prevented other AI researchers to take much of this work seriously [15]. However, Holland developed the modern concept of the genetic algorithm within the framework of machine learning [19], and much more research is still in progress investigating the use of GAs for that purpose, although automatic programming was put aside by researchers for several years. One of the reasons for this was that a conventional GA has some (rather obvious) limitations when used for automatic programming, particularly in terms of representational issues.

Encoding the set of instructions of a programming language and finding a way of combining them in a meaningful manner is not simple by any means, but if a tree structure is used in combination with certain rules that avoid the generation of invalid expressions we can build a primitive parser capable of producing simple programs. This was precisely the approach taken by John R. Koza [23] to develop "genetic programming" in which LISP was originally used to take advantage of the parser built into the language to evaluate the expressions produced.

The tree representation adopted by Koza obviously requires different alphabets and specialized operators to evolve randomly generated programs until they become 100% valid to solve a certain (pre-defined) task, but the underlying principles of the technique can be generalized. Trees are composed of functions and terminals. The functions normally used are the following [23]:

1. Arithmetic operations (e.g., +, -, $\times$, $\div$)

2. Mathematical functions (e.g., sine, cosine, log, exp)

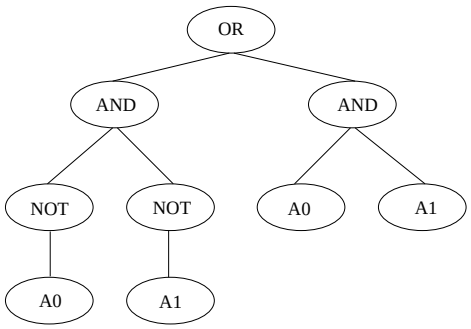

Figure 5: An example of a chromosome in genetic programming.

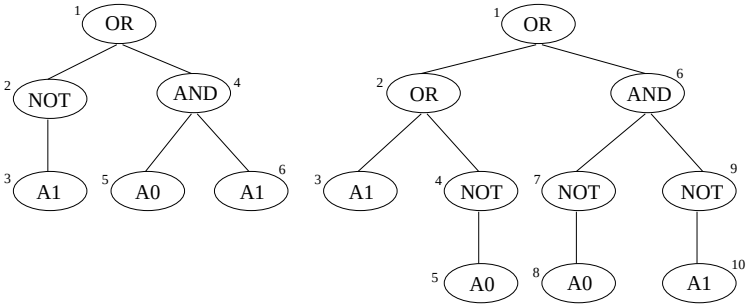

Figure 6: Nodes in the tree are numbered as a previous step to crossover.

3. Boolean operations (e.g., AND, OR, NOT)
4. Conditionals (IF-THEN-ELSE)
5. Iterators (DO-UNTIL)
6. Recursive functions
7. Any other function defined in the current domain

Terminals are typically variables or constants, and can be seen as functions that take no arguments. An example of a chromosome that uses the functions $\mathcal{F}$={AND, OR, NOT} and the terminals $\mathcal{T}$={A0, A1} is shown in Figure 5.

Crossover can be performed by numbering the nodes of the trees corresponding to the 2 parents (see Figure 6) and selecting (randomly) a point in each of them so that the sub-trees below that point are exchanged (see Figure 7, where we assumed that the crossover point for the first parent is 2, and for the second is 6).

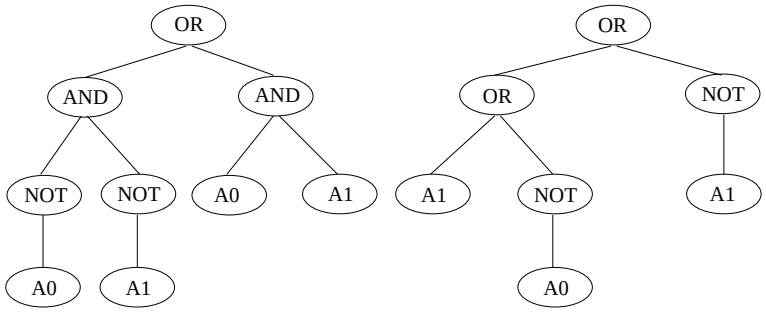

Figure 7: The two children generated after performing crossover.

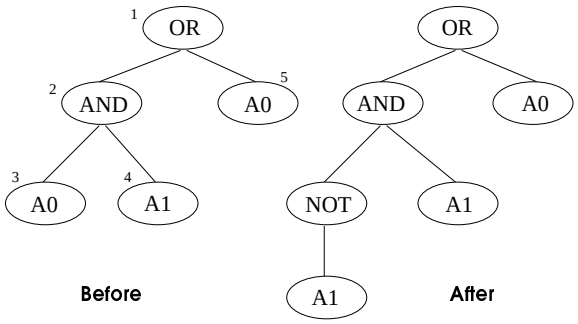

Figure 8: An example of mutation in genetic programming.

Typically, the sizes of the trees of the 2 parents will be different, as shown in the previous example. It should also be observed that if the crossover point happens to be the root of one of the two parent trees, then that entire chromosome will become a sub-tree of the other parent, which is the way of incorporating subroutines into a program. It is also possible that the roots of both parents are selected as the crossover points. In that case, no crossover is performed, and the offspring are the same as their parents.

Normally, the implementation of genetic programming imposes a limit in the maximum depth that a certain tree can reach, to avoid generating (randomly and by using crossover or mutation) trees of considerable size and complexity.

Mutation is performed by selecting (randomly) a certain point in a tree, and then replacing the sub-tree below it with another that is generated randomly. Figure 8 shows an example in which the mutation point is 3.

Permutation is an asexual operator that emulates the inversion operator used in genetic algorithms [17]. It reorders the leaves of a sub-tree after a (randomly) selected point, aiming to strenghten the union of allele combinations with good performance in a chromosome [19]. An example of permutation is shown in Figure 9, where the selected permutation point is 4 (the '*' in-

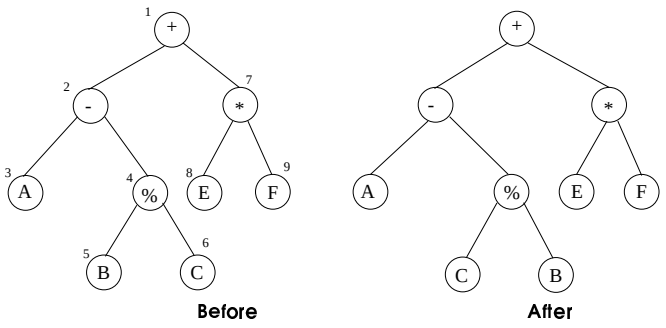

Figure 9: An example of permutation in genetic programming.

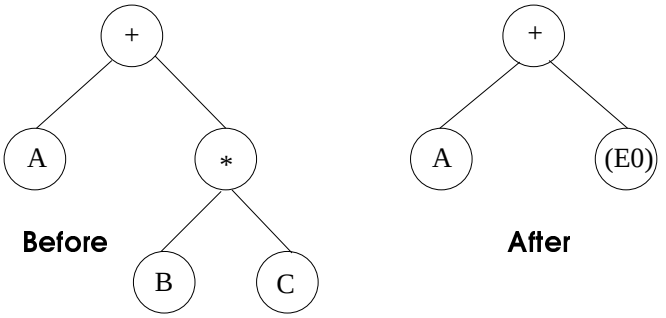

Figure 10: An example of encapsulation in genetic programming.

dicates multiplication, and the '%' indicates "protected division", and it refers to a division operator that avoids that our program crashes when the second argument is zero).

In genetic programming is also possible to protect or "encapsulate" a certain sub-tree that we know is a good building block, to avoid its destruction by any of the genetic operators. The selected sub-tree is replaced by a symbolic name pointing to the actual location of the sub-tree, and the actual sub-tree is compiled separately and linked to the rest of the tree in a way similar to external classes in an object-oriented language. Figure 10 shows an example of encapsulation in which the right sub-tree is replaced by the name **(E0)**.

Normally, is also necessary to do some editing in the expressions generated to simplify them, although the rules for doing that are problem-dependent. For example, if we are generating Boolean expressions, we can apply rules such as the following:

$$(\textbf{AND}\ \mathrm{X}\ \mathrm{X}) \rightarrow \mathrm{X}$$

$$(\textbf{OR}\ \mathrm{X}\ \mathrm{X}) \rightarrow \mathrm{X}$$

$$(\textbf{NOT}\ (\textbf{NOT}\ \mathrm{X})) \rightarrow \mathrm{X}$$

(A2, (B1, (A0, 0, 1), 0), (B0, (B1, 0, 1), (B1, 1, 0)))

C2	C1	C0	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

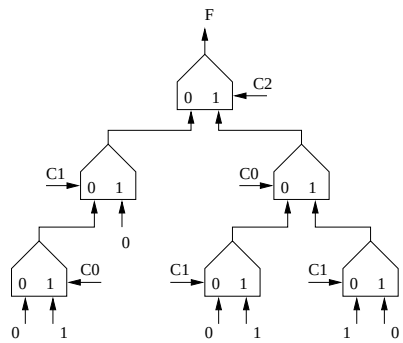

Figure 11: Truth table for logic function specification, the circuit generated, and its coding.

Finally, genetic programming also provides mechanisms to destroy a certain percentage of the population so that we can refresh the chromosomic material after a certain number of generations. This mechanism, called *execution*, is very useful in highly complex domains where our population may not contain a single feasible individuals even after a large number of generations.

In the application reported in this paper, we only used the simplest form of genetic programming: tree encoding with tournament selection (using a tournament size of three), crossover and mutation. None of the other operators previously mentioned were used in our implementation.

6 APPLYING GENETIC PROGRAMMING

There are several issues of interest concerning the use of genetic programming in this domain, and this section aims to help the reader to fully understand the approach taken by the authors by describing the representation and genetic operators used for this work.

The issues taken into consideration for this application and the way in which they were approached are as follows:

- **Representation:** We decided to use binary trees to represent a candidate circuit. Such binary trees are represented by means of lists. Essentially each element of the list is the triplet $(mux, left-child, right-child)$ that encodes subtrees as nested lists. The tree captures the essence of the circuit topology allowing only the children to feed their parent node. In other words, the inputs of a multiplexer can only be chosen from the previous level, as shown in Figure 11.

  Both classes of binary multiplexers (as described in section 4) are implemented, then, for instance, multiplexers $A0$ and $B0$ are activated by the same control bit $C0$ (the least significant bit in Figure 11) but they respond in a particular way to the activation state.

- **Implementation language:** Most of the elements of the population are functional circuits, therefore, they implement (at least partially) the expected logical behavior. Each node of

the tree is a multiplexer whose children could be either another multiplexer or a leaf. The corresponding value of a leaf is, of course, zero or one. Since trees and lists are natural structures in *Prolog* we decided to use this programming language to implement our system so that our implementation could be considerably simpler to develop.

- **Crossover operator:** The exchange of genetic information between two trees is accomplished by exchanging subtrees as shown before. Our implementation does not impose any kind of restriction to the selection of subtrees or crossover points. Node-node, node-leaf, and leaf-leaf exchange are allowed. The particular case when the root node is selected to be exchanged with a leaf is disallowed, so that, no leaf may be mistakenly converted into a node thus avoiding the generation of invalid trees (in such cases the valid children are replicated twice).

- **Mutation operator:** Mutation is implemented in a simple way: first a mutation point is randomly chosen among the nodes and leaves. When a node (multiplexer) is selected, its control input is changed as follows (assuming n control signals): $a_0 \rightarrow a_1$, $a_1 \rightarrow a_2$, $a_{n-1} \rightarrow a_n$, $a_n \rightarrow a_0$. Similarly simple is the mutation of a leaf: $0 \rightarrow 1$, $1 \rightarrow 0$.

- **Fitness function:** The goal is to produce a fully functional design (i.e., one that produces the expected behavior stated by its truth table) which minimizes the number of multiplexers used. Therefore, a fitness function that works in two stages is introduced. At the beginning of the search, only compliance with the truth table is taken into account, and the evolutionary approach is basically exploring the search space. Once the first functional solution appears, we switch to a new fitness function in which fully functional circuits that use less multiplexers are rewarded. Regardless of the current stage of the fitness function, all members of the population have their fitness calculated in every generation. It is the fitness function the only agent responsible for the life span of the individuals.

- **Initial population:** The depth of the trees randomly created for the initial population is set to a maximum value equal to the number n of variables of the logic function (see Section 4). Considering complete binary trees, for n variables, $2^n - 1$ is the upper bound on the number of nodes allowed in the tree. However, we found in our experiments that in the initial population trees of shorter depth were created in larger numbers than trees of greater depth. This led us to adopt a strategy in which the size (depth and number of nodes) of the trees dynamically changed over generations. The goal was to allow our trees to grow without any particular boundaries as to allow a rich phenotypic variation in the population.

7 AN INFORMAL ALGEBRA FOR TERMINAL NODES

The fitness function does not take into account the redundancy of the terminal nodes (nodes whose children are only 0 or 1), that is, identical terminal nodes are pruned away from any solution and counted as just another node.

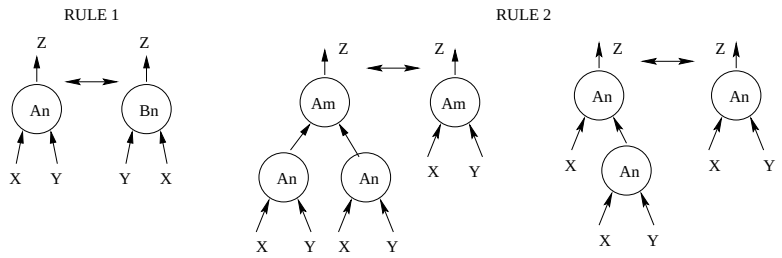

Figure 12: Redundancy and Algebraic Analysis. Node equivalence and subtree equivalence

The function performed by a circuit clearly is a composition of node-functions over the children-node functions, and so on, all the way down to the bottom of the tree. With really low probability either one subtree might be identical to another subtree, or one subtree may consist of the same nodes. Therefore there are some redundant nodes that could be deleted by means of further analysis of the solution delivered by the genetic programming system. We have called it "redundancy and algebraic analysis". Its goal is to improve in at least two ways the best solution found in the last generation. First, the number of elements might be reduced since redundant subtrees (if present) will be removed, and second, it opens the door to implementation decisions, for example, to select the preferred or recommended class of multiplexer to be used.

Terminal nodes are deleted according to the rules shown in Figure 12. Similar rules derived from the problem domain are given in [3]. Rule 1 is applied for transforming one multiplexer class into the other, aiming to maximize redundant nodes that can be deleted and the entire set replaced by just one of them. Subtrees as shown in rule 2 have been observed occasionally. By means of this rule we have been able to reduce the number of nodes of a subtree.

8 COMPARISON WITH STANDARD IMPLEMENTATION

We compare the evolutionary approach against the number of muxes used by a standard multiplexer implementation (see Figure 2). For that sake, several circuits of different degrees of complexity were chosen. The quality of the solutions produced are evaluated using the metric mentioned in Section 3. So far we have limited ourselves to the evolution of circuits with only one output. In all the examples included in this paper, we used a crossover rate of 0.35 and a mutation rate of 0.65 per individual[2]. The population sizes and maximum number of generations used in the following examples were empirically determined.

8.1 Example 1

The first example consists of a function with three inputs and one output, as shown in Table 1. It could be described in words as "the function whose output is 1 when only one of its inputs is 1.

[2]Therefore, probability of mutation per gene=0.65/L, where L is the total number of terminals plus non-terminals in the tree.

C_2	C_1	C_0	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Table 1: Truth table for the first example.

C_3	C_2	C_1	C_0	F
0	0	0	0	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	1	0	1	1
1	1	1	1	1

Table 2: Minterms table for the second example.

Otherwise the output is 0".
Since we have $n = 3$ in this case, theoretically, $2^3 - 1 = 7$ muxes are required. Our approach produces a solution with only 5 muxes. Therefore, we have saved 2 elements (29%). This solution was found using an initial population of 600 elements evolved during 200 generations.

8.2 Example 2

In this second example the problem is to implement a function with four inputs and one output, as shown in Table 2. Only minterms are shown (column F is shown for the sake of clarity).
Since $n = 4$ in this case, then $2^4 - 1 = 15$ muxes are required. The evolutionary approach found the solution shown in Figure 13[3]. This solution only requires 7 muxes, saving then 8 muxes (47%) with respect to the standard implementation. This solution was found using an initial population of 600 individuals evolved during 200 generations.

[3]Only one graphical representation of the solutions produced was included to save space.

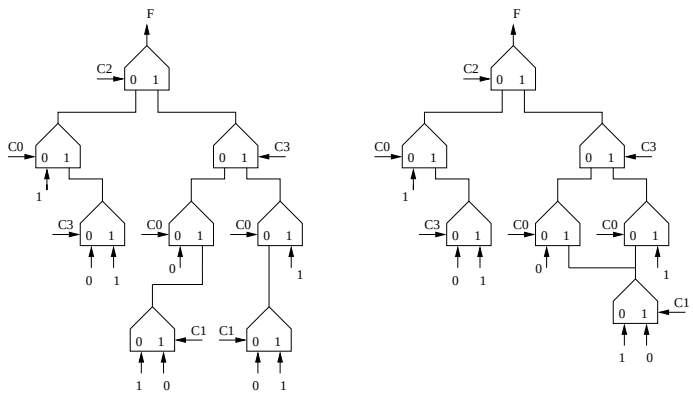

Figure 13: Circuit for the second example and its simplified version after “redundancy and algebraic analysis”.

8.3 Example 3

The third problem is the implementation of a function with 5 inputs and one output. The minterms of the function are the following:

$$f = \sum(0, 1, 3, 6, 7, 8, 10, 13, 15, 18, 20, 21, 25, 26, 28, 30, 31) \tag{2}$$

Since $n = 5$ in this case, then $2^5 - 1 = 31$ muxes are required. The solution found by the evolutionary approach has 15 multiplexers. Therefore, we have saved $31 - 15 = 16$ elements (52%). This solution was found using an initial population of 600 individuals evolved during 1000 generations (due to the increase in complexity of the Boolean function, we had to use a larger number of generations).

8.4 Example 4

In the fourth example we seek a solution for a more complex Boolean function (6 variables). Next is the list of minterms of the function.

$$f = \sum(0, 1, 3, 6, 7, 8, 10, 13, 15, 18, 20, 21, 25, 27, 28, 30, 31, 32, 33, 35, 38, \\ 39, 40, 42, 45, 47, 50, 52, 53, 57, 59, 60, 62, 63)$$

Since $n = 6$ in this case, then $2^6 - 1 = 63$ muxes are required. The solution found by the evolutionary approach has 21 multiplexers. Therefore, we have saved $63 - 21 = 42$ elements (67%). This solution was found using a population of 1500 individuals evolved during 700 generations.

Table 3 summarizes these four examples. The column **n-Mux** shows the number of 1-control line multiplexers needed to implement an n-control lines multiplexer. The column **GP Output** shows the number of nodes in the best solution found by our genetic programming system, and **GP**

Example	Inputs	*n*-Mux	GP Output	GP Algebraic	Total Saved
1	3	7	6	5	2
2	4	15	8	7	8
3	5	31	22	15	16
4	6	63	27	21	42

Table 3: Comparison on the size of the delivered circuits, optimized circuits, and standard implementation.

Algebraic is the further refined solution (the final circuit after manual editing). **Total Saved** is the difference between columns 3 and 5.

9 COMPARING AGAINST OBDDs

The design with multiplexers is akin to the design with binary trees. There does exist, in fact, an isomorphism between trees representing Boolean functions, and the multiplexer-based implementation. Binary trees can be transformed into binary decision diagrams (BDD), BDDs tranformed into Ordered Binary Decision Diagrams (OBDD), and OBDDs into Reduced OBDDs (ROBDDs). ROBDDs provide a canonical representation of logic functions (the diagram representing the function is unique), therefore, they are a powerful tool for reasoning about logic functions. For instance, to determine whether two Boolean functions are identical is equivalent to verify whether their ROBDDs are isomorphic. Properties of this sort are highly cherised, but unfortunately, they can hide the weaknesses of the methods. It is well known that ROBDDs are very sensitive to the variable ordering. For almost any Boolean function, one variable ordering can produce a diagram using an exponential number of nodes, while another ordering can produce a better diagram (sub-exponential number of nodes). Furthermore, the multiplication function does not have any sub-exponential OBDD for any variable ordering [6]. We believe ROBDDs are excellent tools for reasoning about Boolean functions, but they are not suitable for optimal circuit minimization. Many reasearchers have attempted heuristics and algorithms to find the optimum variable ordering that will produce the diagram with the smallest number of nodes. We show three design examples whose solutions are at least as good as the ROBBDs version. The same crossover and mutation rates indicated before were used in these examples.

9.1 Example 5

The first Boolean function we want to synthesize has 6 variables: $F = X_1X_2 + X_3X_4 + X_5X_6$. The OBDD of any function of this sort with n variables has n nodes. The optimal ordering of the variables is $1, 2, 3, 4, 5, 6, \ldots, n$ [6]. We have been able to find optimal solutions to functions with 4, 6, 8 and 10 variables. In Figure 15 the OBDD tree is depicted along with the solution produced by our evolutionary approach. Notice that the tree depth is similar in both designs. The convergence graph corresponding to this problem is shown in Figure 14.

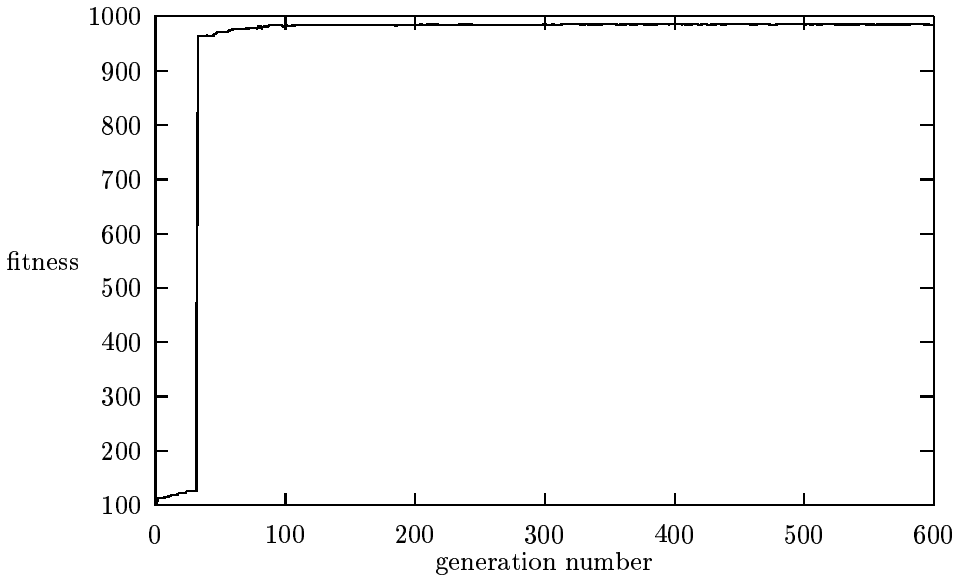

Figure 14: Convergence graph of our GP system when solving the fifth example.

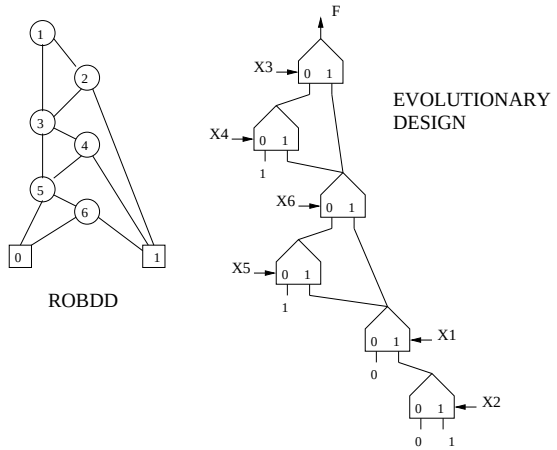

Figure 15: Synthesis of Example 5

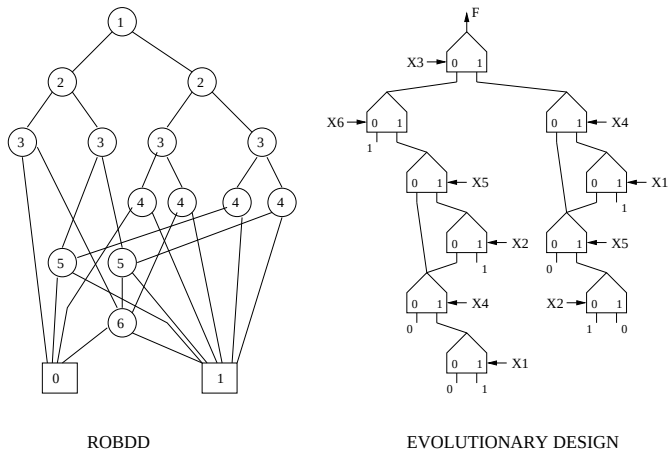

Figure 16: Synthesis of Example 6

Our genetic programming system found the optimal solution at generation 300 using a population size=990 individuals.

9.2 Example 6

The next design is the synthesis of a similar function with 6 variables: $F = X_1X_4 + X_2X_5 + X_3X_6$. The optimal solution found by OBDDs to this problem has 14 non-terminal nodes with variable ordering $1, 2, 3, 4, 5, 6$. Thus, no other variable ordering will find a better solution using OBDD techniques [6]. In Figure 16 we show the evolved optimal solution delivered by the genetic programming system. It is implemented with only 10 nodes.

Our genetic programming system found the optimal solution at generation 219 using a population size=990 individuals.

9.3 Example 7

The "odd parity" function is a very hard problem to solve using multiplexers and genetic programming. In fact we have found the optimal solution only for problems having four or fewer variables. The difficulty is due in part to the fact that the ideal solution requires only XOR gates. Therefore, any approach for which XOR gates cannot be mapped one-to-one into another set of primitives will have more elements than the number of XORs. One single Mux cannot implement an XOR gate, but using Muxes as the circuit primitive, the genetic programming system has proven able to optimally implement the required function.

The Boolean function of 3 variables is: $F = X_1 \oplus X_2 \oplus X_3$ (where $\oplus$ refers to the XOR function). Using OBDDs, the solution for n variables has at most $2n - 1$ non-terminal nodes. In Figure 17 we show the OBDD solution, and the evolved optimal solution delivered by our genetic programming system with 7 nodes that can be reduced to 5.

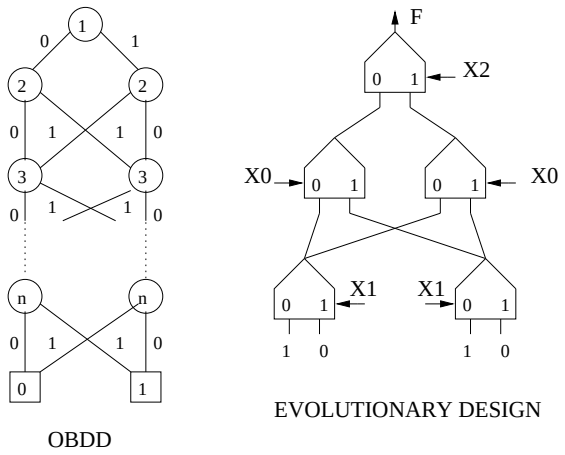

Figure 17: Synthesis of Example 7

Our genetic programming system found the optimal solution at generation 7 using a population size=990 individuals.

10 CONSISTENCY OF THE CONVERGENCE

When using Evolutionary Computation techniques it is indispensable to evaluate the consistency of the method. Thus, one can be confident that: a) the solution to new problems could be found, and b) current solutions were not found by mere chance of a combination of population size and crossover and mutation parameters of the algorithm.

10.1 Example 8

The following problem is inspired in what now is known as the *11-multiplexer* and *20-multiplexer* problems. Droste [14] has shown a partially specified function approach to these problems. We wish to verify the ability of the system for designing Boolean functions with a "large" number of arguments and specific topology. That is, the topology is preserved as the number of variables increases. Boolean functions with 2^k variables (where $k = 1, 2, \ldots$), are implemented with exactly $(2 \cdot 2^k) - 1$ binary muxes. For example, for $k = 3$, a Boolean function of $2^3 = 8$ variables is implemented with exactly 15 muxes when the truth table is specified as shown in Table 4. For any k (i.e., the number of variables), we specify the table in a similar way. Notice that there are exactly $2 \cdot 2^k + 2$ entries in the table.

Table 5 shows the high rate of convergence of the GP system to the optimum. We ran 100 experiments for each function (each k). The column **vars** shows the number of variables for some integer k, **muxes** refers to the optimum number of binary muxes needed to implement the partial Boolean function, and **average** indicates the average number of iterations needed to find the optimum. In all

$\mathbf{X_7}$	$\mathbf{X_6}$	$\mathbf{X_5}$	$\mathbf{X_4}$	$\mathbf{X_3}$	$\mathbf{X_2}$	$\mathbf{X_1}$	$\mathbf{X_0}$	**F**
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	1	0	1
0	0	0	0	0	1	0	0	1
0	0	0	0	1	0	0	0	1
0	0	0	1	0	0	0	0	1
0	0	1	0	0	0	0	0	1
0	1	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	1
0	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1
1	1	1	1	0	1	1	1	1
1	1	1	1	1	0	1	1	1
1	1	1	1	1	1	0	1	1
1	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	0

Table 4: Partially specified function of Example 8 ($k = 3$)

k	vars	muxes	average	confidence
2	4	7	60	$> 90\%$
3	8	15	200	$> 90\%$
4	16	31	700	$> 90\%$

Table 5: Average number of iterations needed for solving Example 8

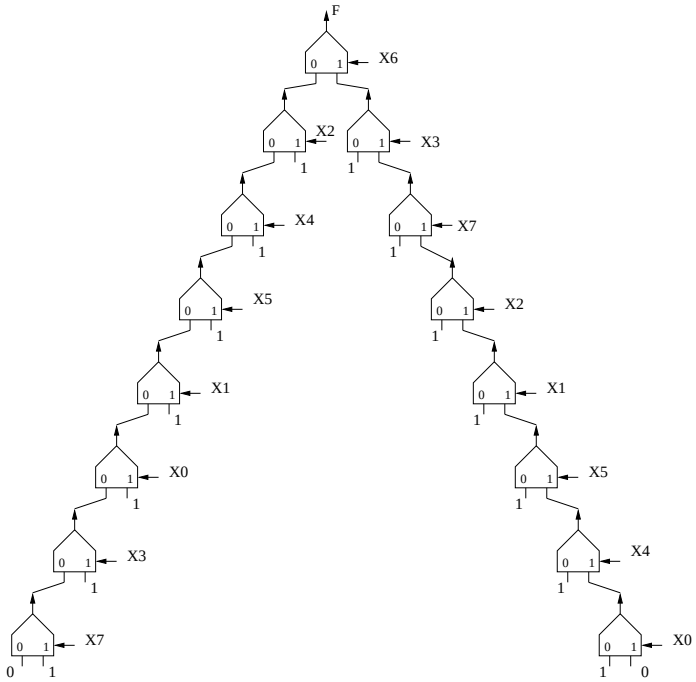

Figure 18: Synthesis of Example 8 ($k = 3$)

cases, we found optimum size circuits in more than 90% of the iterations. The topology of these circuits for any k is similar to that of the Figure 18.

CONCLUSIONS

Human designers seem to prefer top-down methods since the nature of the design process becomes the simplification and reduction of an initial circuit into a minimum form. That is, a solution exists in the very first step of a top-down process. Such is the case of OBDDs based techniques. It seems natural to human beings to work with schemes of this sort. Synthesis develops in the opposite direction. It denotes a creative process that begins with a functional specification (truth table), and ends with the architecture that reproduces the required behavior (the circuit). Circuits designed in this way tend to be somewhat different to the usual (human) solutions. A plausible explanation is

that synthesis methods explore regions of the space of solutions that top-down methods never reach during the refinement process. The circuits of the examples 5,6 and 7 show that characteristic: the node ordering is not only different but it is (mostly) unknown to human designers. Better yet, the circuits generated have optimal size. In example 7 ("odd-parity function"), it is also interesting to note that a basic building block is determined by the GP system. By replicating the multiplexer-based elementary module, designers can define any odd-parity function of n variables (a procedure similar to the replication of the XOR module).

We have shown evidence of similar ability of OBDDs and Genetic Programming for circuit design. Nevertheless, the many years of effort in deriving OBDDs simplification rules do not outperform a heuristic method with no knowledge of the problem domain.

11 ACKNOWLEDGMENTS

The first author acknowledges support from CONCyTEG through project 03-02-K118-037 part 2, the second author acknowledges support from NSF-CONACyT through project number 32999-A.

References

[1] Arturo Hernández Aguirre, Bill P. Buckles, and Carlos A. Coello Coello. Circuit Design using Genetic Programming: An Illustrative Study. In *Proceedings of the 10th NASA Symposium on VLSI Design*, pages 4.1.1–4.1.9. University of New Mexico Press, The University of New Mexico, Albuquerque, New Mexico, march 2002.

[2] Arturo Hernández Aguirre, Carlos A. Coello Coello, and Bill P. Buckles. A Genetic Programming Approach to Logic Function Synthesis by means of Multiplexers. In D. Keymeulen A. Stoica and J. Lohn, editors, *The First NASA/DoD Workshop on Evolvable Hardware*, pages 46–53. IEEE Computer Society, 1999.

[3] S. B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C–27(6):509–516, June 1978.

[4] A. E. A. Almaini, J.F. Miller, and L. Xu. Automated Synthesis of Digital Multiplexer Networks. *IEE Proceedings Pt E*, 139(4):329–334, July 1992.

[5] J. Wirt Atmar. *Speculation on the Evolution of Intelligence and Its Possible Realization in Machine Form*. PhD thesis, New Mexico State University, Las Cruces, New Mexico, 1976.

[6] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transaction on Computers*, C–35(8):677–691, August 1986.

[7] R. E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computer Surveys*, 24(3):293 – 318, September 1992.

[8] Carlos A. Coello, Alan D. Christiansen, and Arturo Hernández Aguirre. Using Genetic Algorithms to Design Combinational Logic Circuits. In Cihan H. Dagli, Metin Akay, C. L. Philip Chen, Benito R. Farnández, and Joydeep Ghosh, editors, *Intelligent Engineering Systems Through Artificial Neural Networks. Volume 6. Fuzzy Logic and Evolutionary Programming*, pages 391–396. ASME Press, St. Louis, Missouri, USA, nov 1996.

[9] Carlos A. Coello Coello, Alan D. Christiansen, and Arturo Hernández Aguirre. Automated Design of Combinational Logic Circuits using Genetic Algorithms. In D. G. Smith, N. C. Steele, and R. F. Albrecht, editors, *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms*, pages 335–338. Springer-Verlag, University of East Anglia, England, April 1997.

[10] Carlos A. Coello Coello, Alan D. Christiansen, and Arturo Hernández Aguirre. Use of Evolutionary Techniques to Automate the Design of Combinational Circuits. *International Journal of Smart Engineering System Design*, 2(4):299–314, June 2000.

[11] M. Davio, J. P. Deschamps, and A. Thayse. *Digital systems, with algorithm implementation.* Wiley, New York, USA, 1983.

[12] Hugo de Garis. Evolvable Hardware: Genetic Programming of a Darwin Machine. In Colin Reeves, R. F. Albrecht, and N. C. Steele, editors, *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms*, pages 117–123, Inssbruck, Austria, 1993. Springer-Verlag.

[13] Rolf Drechsler, Nicole Göockel, and Bernd Becker. Learning Heuristics for OBDD Minimization by Evolutionary Algorithms. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature IV. Lecture Notes in Computer Science 1141*, pages 730–739, Berlin, Germany, September 1996. Springer-Verlag.

[14] Stefan Droste. Efficient Genetic Programming for Finding Good Generalizing Boolean Functions. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Proceedings of the Second Annual Conference on Genetic Programming*, pages 82–87. Morgan Kaufmann Publishers, San Francisco, California, July 1997.

[15] David B. Fogel. *Evolutionary Computation. Toward a New Philosophy of Machine Intelligence.* The Institute of Electrical and Electronic Engineers, New York, 1995.

[16] George J. Friedman. Selective Feedback Computers for Engineering Synthesis and Nervous System Analogy. Master's thesis, University of California at Los Angeles, February 1956.

[17] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing, Reading, Massachusetts, 1989.

[18] R.K. Gorai and A. Pal. Automated synthesis of combinational circuits by cascade networks of multiplexers. *IEE Procedings Pt E*, 137(2):164–170, March 1990.

[19] John H. Holland. *Adaptation in Natural and Artificial Systems. An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, Cambridge, Massachusetts, 1992.

[20] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning about systems.* Cambridge University Press, 2000.

[21] Hitoshi Iba, Masaya Iwata, and Tetsuya Higuchi. Gate-Level Evolvable Hardware: Empirical Study and Application. In Dipankar Dasgupta and Zbigniew Michalewicz, editors, *Evolutionary Algorithms in Engineering Applications*, pages 260–275. Springer-Verlag, Berlin, 1997.

[22] Hiroaki Kitano and James A. Hendler, editors. *Massively Parallel Artificial Intelligence.* MIT Press, 1994.

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

[24] John R. Koza, David Andre, III Forrest H. Bennett, and Martin A. Keane. Use of automatically defined functions and architecture-altering operations in automated circuit synthesis with genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Proceedings of the First Annual Conference on Genetic Programming*, pages 132–140, Cambridge, Masachussetts, July 1996. Stanford University, The MIT Press.

[25] John R. Koza, III Forrest H. Bennett, David Andre, and Martin A. Keane. Automated WYWIWYG design of both the topology and component values of electrical circuits using genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Proceedings of the First Annual Conference on Genetic Programming*, pages 123–131, Cambridge, Masachussetts, July 1996. Stanford University, The MIT Press.

[26] Glen G. Langdon. A decomposition chart technique to aid in realizations with multiplexers. *IEEE Transactions on Computers*, C–27(2):157–159, February 1978.

[27] Sushil J. Louis. *Genetic Algorithms as a Computational Tool for Design.* PhD thesis, Department of Computer Science, Indiana University, August 1993.

[28] Sushil J. Louis and Gregory J. Rawlins. Using genetic algorithms to design structures. Technical Report 326, Computer Science Department, Indiana University, Bloomington, Indiana, February 1991.

[29] J. F. Miller, P. Thomson, and T. Fogarty. Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study. In D. Quagliarella, J. Périaux, C. Poloni, and G. Winter, editors, *Genetic Algorithms and Evolution Strategy in Engineering and Computer Science*, pages 105–131. Morgan Kaufmann, Chichester, England, 1998.

[30] Ajit Pal. An algorithm for optimal logic design using multiplexers. *IEEE Transactions on Computers*, C-35(8):755–757, August 1986.

[31] Hidenori Sakanashi, Tetsuya Higuchi, and Yukinori Kakazu. Evolution of Binary Decision Diagrams for Digital Circuit Design using Genetic Programming. In T. Higuchi, M. Iwata, and L. Weixin, editors, *Proceedings of the First International Conference, ICES'96, Tsukuba, Japan*, pages 470–481. Springer Verlag, Heidelberg, Germany, October 1997.

[32] Christoph Scholl and Bernd Becker. On the Generation of Multiplexer Circuits for Pass Transistor Logic. In *Proceedings of Design, Automation, and Test in Europe*, page 372, Paris, France, March 2000. IEEE Computer Society.

[33] Claude E. Shannon. The Synthesis of Two-Terminal Switching Circuits. *Bell System Technical Journal*, 28(1):59–98, 1949.

[34] Sajjan G. Shiva. *Introduction to Logic Design*. Scott, Foresman and Company, 1988.

[35] Adrian Thomson, I. Harvey, and Philip Husbands. Unconstrained evolution and hard consequences. In E. Sanchez and M. Tomassini, editors, *Toward Evolvable Hardware (Lecture Notes in Computer Science, Vol. 1062)*, pages 136–165, Heidelberg, Germany, 1996. Springer-Verlag.

[36] A.J. Tosser and D. Aoulad-Syad. Cascade networks of logic functions built in multiplexer units. *IEE Proceedings Pt E*, 127(2):64–68, March 1980.

[37] Masayuki Yanagiya. Efficient genetic programming based on binary decision diagrams. In Toshio Fukuda and Takeshi Furuhashi, editors, *Proceedings of the 1995 IEEE International Conference on Evolutionary Computation*, pages 234–239, Nagoya, Japan, 1995. IEEE.

[38] Stephen S. Yau and Calvin K. Tang. Universal Logic Modules and Their Application. *IEEE Transactions on Computers*, C–19(2):141–149, February 1970.