Journal of Circuits, Systems, and Computers

SAT Based Fitness Scoring for Digital Circuit Evolution

Mumtaz Ali and Osman Hasan
School of Electrical Engineering and Computer Science
National University of Sciences and Technology (NUST)
Sector H-12, Islamabad, Pakistan
{*10mseemali,osman.hasan*}*@seecs.nust.edu.pk*

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Evolutionary computation uses Darwinian principles to find solutions from a given search space and forms the basis for evolving digital circuits. One of the most computationally expensive steps in evolutionary computation is the comparison of the candidate circuit or chromosome with the target truth table. We propose to use a satisfiability solver, to improve upon the efficiency of this process, which is traditionally done using exhaustive simulation. The paper presents an implementation of the satisfiability solver, which is in turn used to develop a digital circuit evolution methodology based on the principles of cartesian genetic programming. The proposed methodology performs better, in terms of speed of design space exploration, for circuits whose behavior can be expressed compactly in terms of conjunctive normal form clauses. For illustration purposes, the proposed methodology has been used to evolve various commonly used digital circuits and a few benchmarks.

Keywords: Cartesian Genetic Programming, Digital Circuit Evolution, Evolutionary Computation, Satisfiability Solvers, Formal Verificaiton

1. Introduction

Evolvable Hardware is a domain in which evolutionary computation or other bio-inspired algorithms are used for automated hardware design, dynamic adaptive hardware, self replication or self repair [10,18,30]. This makes evolvable hardware a very interesting multidisciplinary research domain involving biology, computer science and engineering. The focus of this paper is on using the foundations of evolvable hardware in the context of automated combinational digital circuit design, i.e, a scenario in which evolutionary algorithms are used to evolve combinational digital circuits. This approach is known for providing better synthesis results than the traditional methods [4,6].

Figure 1 depicts the genetic programming based search methods, which are very commonly used for digital circuit evolution. The main strength of genetic programming is the ability to automatically search the required solution from a given search space without any prior knowledge about the problem. The input to

the genetic programming based evolutionary computing is a target truth table and its goal is to find a solution circuit, or *chromosome*, whose truth table matches with the truth table of the target function.

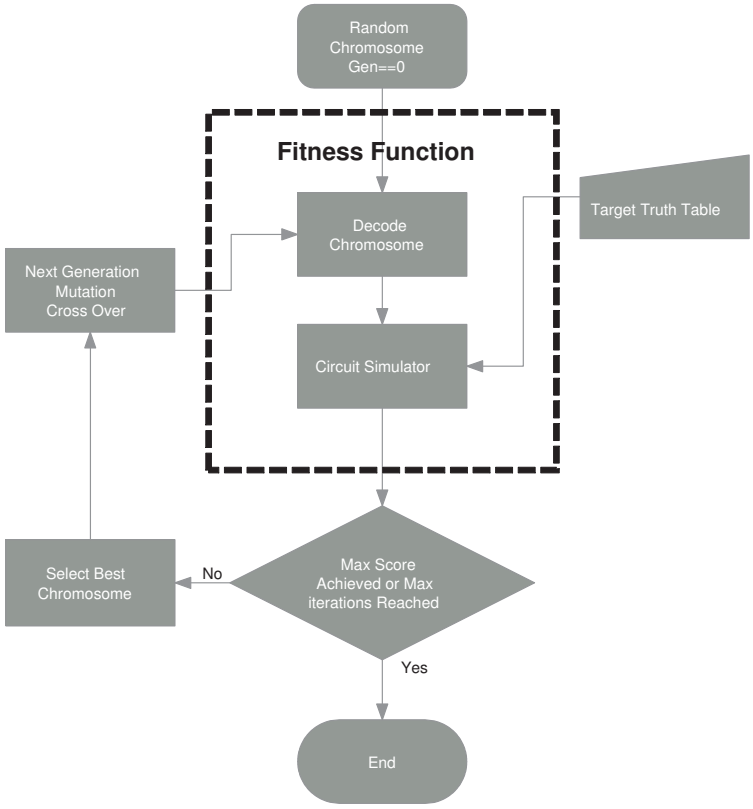


Fig. 1: Genetic Programming based Search Methods.

Initially, the algorithm starts with a population of random chromosomes. The algorithm has access to a *fitness function* that measures the closeness level of a candidate chromosome and the target function. This fitness function is used to evaluate each chromosome of the present generation and assign a fitness score to it. Chromosomes with the best fitness score are selected to produce the next generation of population by recombination/crossover or random mutation. A number of selection strategies have been reported in the open literature. For example, in the tournament selection strategy, chromosomes are randomly divided into groups and the chromosomes with the best fitness score from each group are selected to produce the new generation [8]. The algorithm keeps on running until the target solution is achieved, or the maximum number of allowed iterations is reached.

A number of genetic programming based evolutionary techniques for evolving digital circuits have been reported in the literature. Some promising ones include Koza's genetic programming [16] and cartesian genetic programming (CGP) [21]. Cartesian genetic programming is more popular mainly because it is more efficient in terms of computation time and required resources than the other biologically-inspired methods [22,26,29].

Traditionally, the fitness scores are calculated using exhaustive simulation in digital circuit evolution. The main idea is to compare the output of the given chromosome and the target function using all the possible input patterns. This kind of exhaustive simulation consumes a significant amount of computation time, which grows exponentially with an increase in the number of inputs or functional complexity of the target function. This enormous computation time requirement is one of the major factors that limits the scope of digital circuit evolution [42].

In this paper, we propose to solve the above-mentioned computation time problem by a methodology that allows to use #SAT solvers [34] to assess the fitness scores in digital circuit evolution. Traditional SAT solvers are known to be computationally faster than simulation in the task of equivalence checking [2,5]. However, traditional SAT solvers just provide a Yes/No answer for the satisfiability of a Boolean expression and cannot provide a comparison score between the circuits that are being checked for equivalence, which is the main requirement in the context of digital circuit evolution. #SAT solver provide this capability and thus are used in this paper. To the best of our knowledge, both SAT and #SAT solvers have never been used for fitness scoring in digital circuit evolution before.

The proposed methodology is primarily based on the CGP technique [23]. We have developed a variant of CGP in which chromosomes are evaluated using #SAT solving. We implemented our own #SAT solver instead of using an off-the-shelf one in order to avoid the interfacing overhead required for connecting one of the available #SAT solvers with the proposed methodology. This way, we can do both conjunctive normal form (CNF) reduction and use #SAT in two steps within the same C program. On the other hand, opting for an existing #SAT solver would have required the output of CNF reduction to be translated to the reduced form compatible with the format of the off-the-shelf #SAT solver. Thus, the current implementation exhibits better performance compared to the choice with an external #SAT solver.

The main principle of the proposed methodology is to convert the target function behavior and the given chromosome into a CNF and then evaluate their fitness based on the number of inputs assignments for which the CNF is unsatisfiable. The complete code is written in C++. For illustrating the effectiveness of the proposed methodology and our development, we utilize it to evolve digital circuits exhibiting the behavior of adders, multipliers, multiplexers, even parity circuits, encoders and a few LGSynth91 benchmarks [19]. It is worth mentioning that a significant time gain was observed for circuits in which the number of CNF clauses in the CNF representation exhibit a linear dependance on the number of inputs.

2. Related Work

Scalability is a two-fold problem in the domain of digital circuit evolution [28,42]. Firstly, the evaluation of the closeness of a candidate solution with the target solution is not scalable. Secondly, the search space grows exponentially with respect to the size of the problem and thus is not easy to handle with the given computation and memory constraints. Various methods have been proposed to tackle these scalability issues and this section summarizes them besides providing some related work for SAT solving based equivalence checking.

2.1. *Scalability of Representation*

A search space in circuit evolution is the space of all possible representations of circuits in a chromosome. Thus, the size of the search space is directly related to the length of the chromosome, which is chosen by the designer based on the size of the circuit. As the circuit size grows, the chromosome size has to grow accordingly, which results in an exponential growth in the size of the search space. This in turn increases the computation requirements for finding the solution circuit. Numerous researchers have tried to alleviate this problem. The authors in 25, 28 have proposed to perform digital circuit evolution at the functional level and thus use large building blocks instead of gates as basic components. Some limitations of this approach include the manual definition of such blocks for every design by the designer and the inefficiency of digital circuit evolution algorithms at this higher abstraction level with more complex functionality.

Incremental evolution is another option in which a problem is first divided into sub modules or units [33,36,37]. First, evolution is performed to individually evolve these units and then these units act as building blocks for evolution of more complex circuits. In the modular approach, the modules are automatically defined to be reused in the evolution process [14,44]. Combination of modular and incremental evolution has also been used [7,31]. Computational developments have brought a new direction to this field with promising theoretical and practical results, but the issue of scalability is still an open challenge because all the above mentioned techniques compromise on the diversity of the search space [9,11,12,17,20,45] .

2.2. *Scalability of Evaluation Time*

Fitness evaluation time also grows exponentially with an increase in the size of circuit. Traditionally, fitness score is calculated using exhaustive simulation where all possible inputs are tested. There are a number of strategies reported so far to tackle this issue. In some cases such as filters, classifiers or robot controllers, where a circuit is required to work only for a subset of inputs, fitness scoring is done by partial simulation. However, this approach is not applicable to circuits where a correct response is required for all possible inputs [13]. If a target system is linear, it is possible to completely evaluate a candidate circuit using only one input vector [41].

Z. Vasicek et al. proposed to verify a candidate circuit against the target solution using a SAT based equivalence checking algorithm [39]. Due to limitation of traditional SAT solvers (which work in yes/no fashion), the method only works for the post synthesis optimization phase where a circuit is initially synthesized using a conventional synthesis tool first and then it is further optimized in evolutionary framework using SAT solvers [39]. In the current paper, we overcome this restriction by proposing a #SAT solver based fitness scoring.

Based on a similar motivation like ours, another approach to facilitate the evolutionary circuit design process for complex circuits, especially arithmetic circuits has been recently proposed [40]. The candidate circuit is represented as a binary decision diagram (BDD) and its fitness is evaluated based on the Hamming distance with the given specification, also represented as a BDD. This kind of fitness evaluation has been shown to be more faster than rigorous simulations for all possible the inputs and more accurate. The proposed approach is quite similar to this previous work, but instead of using the BDDs for calculating the Hamming distance between the candidate and specified circuits, we propose to do this using a #SAT solver.

Both BDDs and SAT solvers have been widely used for satisfiability and validity checks for propositional logic formula and they have their own strengths and weaknesses. Thus, the proposed approach can be considered as a complementary approach to the work presented in 40. A very natural future direction of our work is to provide the support of both BBD and #SAT solver to the evolutionary circuit design process so that they can be used to complement each others weaknesses.

2.3. *SAT Based Formal Verification*

The main focus of SAT solving is to optimize the search for a satisfying assignment of the variables of a given boolean function in terms of time taken and resources utilized. Growing demand for more efficient and scalable verification solutions have fuelled the research in SAT based verification techniques in the last two decades [2,5,15,27,35]. MiniSAT [1] and Picosat [3] are two of the most popular SAT solvers these days.

Converting the boolean expression into their corresponding CNF format is an important step in SAT based equivalence checking algorithms. The main challenge here is to obtain the most compact CNF form, i.e., with the least number of clauses, since the number of clauses not only affects the memory consumption but also the performance of a SAT solver in terms of computation time. Tseitin proposed a very promising algorithm for this purpose, which uses new variables for all intermediate nodes of the circuit instead of resolving them [38]. The addition of new variables keeps the number of clauses linear with the size of circuit.

Many extensions of Tsetin's classical algorithm have been proposed in the literature, e.g. 43. However, in our case we cannot use Tseitin's algorithm because we need to calculate all possible assignments in terms of primary inputs and the new variables for intermediate nodes do not allow this. We therefore use the basic

Table 1: List of Cell Functions in CGP.

Function Number	**Function**	**Function Number**	**Function**
0	0	10	xor(a,b)
1	1	11	xor(a,!b)
2	a	12	or(a,b)
3	b	13	or(a,!b)
4	not(a)	14	or(!a,b)
5	not(b)	15	or(!a,!b)
6	and(a,b)	16	mux(a,b)
7	and(a,!b)	17	mux(a,!b)
8	and(!a.b)	18	mux(!a,b)
9	and(!a,!b)	19	mux(!a,!b)

method of CNF conversion in the proposed methodology.

3. Preliminaries

In this section, we describe some foundational concepts about digital circuit evolution and SAT based equivalence checking along with some commonly used terminology.

3.1. *Digital Circuit Evolution*

The main principle of digital circuit evolution is based on the genetic programming based evolutionary computing as illustrated in Fig. 1. The target specification is the functionality of the desired circuit in this case and the chromosomes represent digital circuits in coded form. The fitness scores are usually computed by comparing each chromosome with the target circuit using exhaustive simulation.

The CGP technique, developed by Miller and Thomson, is one of the most widely used techniques used for the digital circuit evolution [21,22,24,26,29]. In CGP, we represent a candidate circuit as a two dimensional (n_r x n_c) grid of programmable nodes where n_c represents the number of columns and n_r represents the number of rows. Each node in this 2D grid is programmable and can acquire any one of the 20 available functions, which are given unique identifiers in Table 1, or any other similar functions if allowed.

We illustrate the CGP based evolution of digital circuits by considering an example of a candidate circuit given in Fig. 2. This circuit is composed of a 2x2 grid with four nodes, three inputs and two outputs. In CGP, the inputs and outputs of the nodes are represented by distinct integers, such that the inputs are labeled first, starting from the integer 0, and then the output of each node is labeled in a column wise fashion. Thus, in our example circuit of Fig. 2, numbers 0,1 and 2 represent the inputs and 3,4,5 and 6 represent the outputs of nodes, respectively.

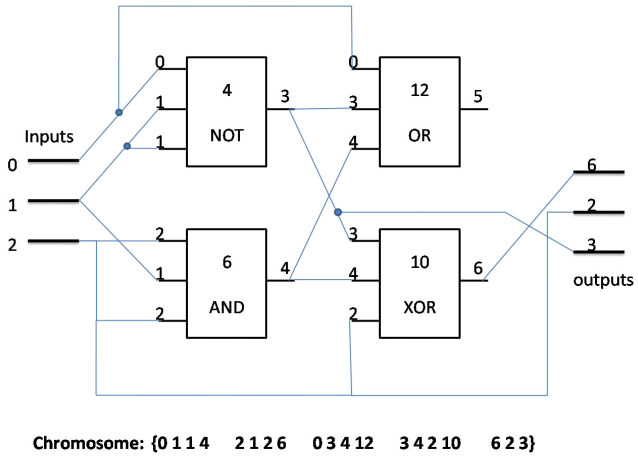


Fig. 2: A Candidate Circuit.

The input nodes can be connected either to a node output from one of the previous ℓ columns or to one of the inputs of the circuit. This way feedback loops are restricted. The factor ℓ denotes *level-back* which is used to limit the design space by not allowing a node input to be connected to any column which is behind ℓ allowed columns. It is important to note that each node consists of three inputs as the cell functions 16-18, given in Table 1, require three inputs. Thus, in order to have the flexibility to adapt any cell functionality the maximum number of inputs is used for every node.

A chromosome representation of the candidate circuit of Fig.2 is also provided below it in the form of an integer string. A node in the chromosome is represented by four integers. First three integers show the input connectivity of the node and the last integer identifies its functionality, as given in Table 1. The last set of integers in the chromosome shows the output connectivity and thus contains integers equal to the number of outputs. This entry, in the case of our example, is 623, which corresponds to the three outputs of the candidate circuit. In this way, the chromosome captures the complete behavior, including the functionality and structure, of the candidate circuit.

Initially CGP starts with a population of randomly chosen chromosomes, which is usually referred to as the first generation. Fitness function evaluates each chromosome and assigns a fitness score to it. The chromosome with the best fitness score is selected as the parent for the next generation. The selected chromosome is then randomly mutated to produce the next generation. Random mutation is a process in which connectivity of the node input or output is randomly changed while the mutation rate is determined by the user.

The parameter λ, which is a user defined term, defines the size of the population in one generation. Each new generation includes the best chromosome from

the previous generation and its λ mutated versions. CGP keeps evolving new generations until the solution circuit is acquired or the maximum number of iterations is reached.

The fitness function plays the most important role in the evolution of digital circuits as it is the fitness score that guides the selection of the next generation. In CGP, fitness of a chromosome is calculated using exhaustive simulation as illustrated in Fig.3. The main idea is to apply all possible inputs to the candidate circuit and compute the Hamming distance between each one of its outputs with the corresponding output of the truth table of the target circuit. The Hamming distance is then used to assess the fitness score of the candidate circuit. This process involves testing for all possible inputs and thus can be quite expensive in terms of computation time. This fact is one of the main limiting factors in the domain of digital circuit evolution.

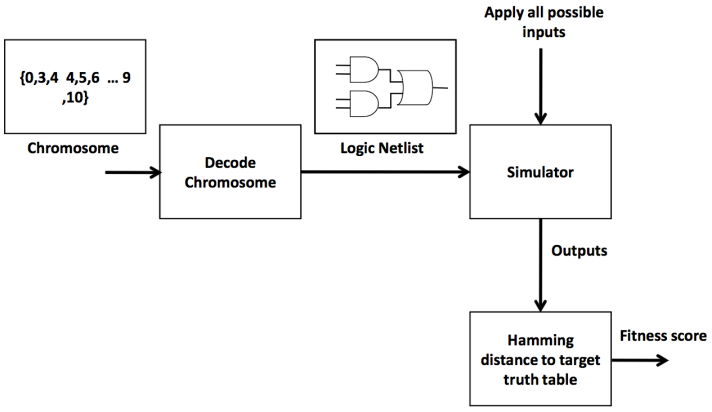


Fig. 3: Fitness Function of Conventional CGP.

Parallel simulation is a technique that can be used to improve the fitness score calculation time for standard CGP [21]. The main principle of parallel simulation is to leverage upon the bitwise logical operations supported by languages like C. This allows us to perform more than one evaluation of a circuit by a single instruction. For example, an integer in C has a width of 32-bits so 32 logical operations for a gate can be executed by one instruction. If we consider the basic equality: 2^5=32, we can simulate a circuit with 5 inputs by applying a single 32 bit vector at each input. The current paper is also targeted towards the same goal, i.e., improving the fitness scoring time. However, we propose to use SAT solver based scoring instead of exhaustive simulation. The proposed method performs better than the parallel simulation based fitness scoring, which is the state-of-the-art technique, for a wide range of circuits, as will be demonstrated in Section 6.

3.2. *Functional Equivalence Checking using SAT*

Functional equivalence checking is a method in which two different structures are verified to be functionally equivalent and it is commonly used in logic synthesis verify that the synthesized netlist is functionally verified against the reference circuit [2].

A SAT solver is an implementation of an algorithm for the SAT problem, which is the problem of determing if a given boolean expression is satisfiable, i.e., it is true atleast for one particular assignment of its variables [1,3]. SAT solvers have found an enormous application in equivalence checking of boolean circuits because it can handle many interesting equivalence checking problems automatically.

The main idea is to form the XOR function of the two boolean expressions of the digital circuits, whose equivalence needs to be verified, in the conjunctive normal form (CNF), i.e., a Boolean formula composed of a conjunction of clauses where each clause is formed by a disjunction of literals that represent Boolean variables. A SAT solver is used to check the satisfiability of the resulting CNF and the two circuits are termed functionally equivalent if and only if the CNF is unsatisfiable, i.e, the XOR of the two outputs is never true for any input variable assignment.

Algorithm 1 briefly explains the usage of SAT solvers for functional equivalence checking. A mitter, used on the line 2 of Algorithm 1, represents the bit-wise XOR operation between the outputs of the two circuits and thus is false in case the circuits are equivalent. Each mitter is then converted to its CNF format and is then passed to the SAT solver to check if it is satisfiable for any assignment of input variables. In case a mitter is found to be satisfiable, circuits are not equivalent to one another and the satisfying assignment can be used for debugging. A modern SAT solver provides a more efficient way to search for a satisfying assignment than exhaustive simulation and therefore outperforms it [32,39].

Algorithm 1 SAT Based Equivalence Checking.

Inputs:
CircuitA: a set of functions $\{y_1, y_2, ...y_N\}$
CircuitB: a set of functions $\{f_1, f_2, ...f_N\}$
Output:
A satisfiable assignment

$M \leftarrow NULL$
for $1 = 1 \rightarrow N$ **do**
  $M \leftarrow M \vee (y_i \oplus f_i)$ ▷ M is a mitter
end for
$CNF \leftarrow boolean\_logic\_to\_CNF(M)$
$\{sat, assignment\} \leftarrow satsolver(CNF)$
if sat is true **then**
  Print("circuits are not equal for assignment:")
  Print(assignment)
else
  Print("CircuitA and CircuitB are functionally equivalent")
end if

Traditional SAT solvers provide a Yes/No answer, i.e., they can merely inform us if a logical formula is satisfiable or not. They lack the ability to find the closeness of two circuits, i.e., fitness scores, which is the main requirement in the case of digital circuit evolution. Unlike a SAT solver, #SAT solver has the capability to provide the total count of satisfiable assignments for a given logical formula. Therefore, we propose to use #SAT solvers for fitness scoring.

4. Proposed Methodology

Figure 4 presents a general block diagram of the proposed methodology, which is primarily based on CGP and #SAT solving.

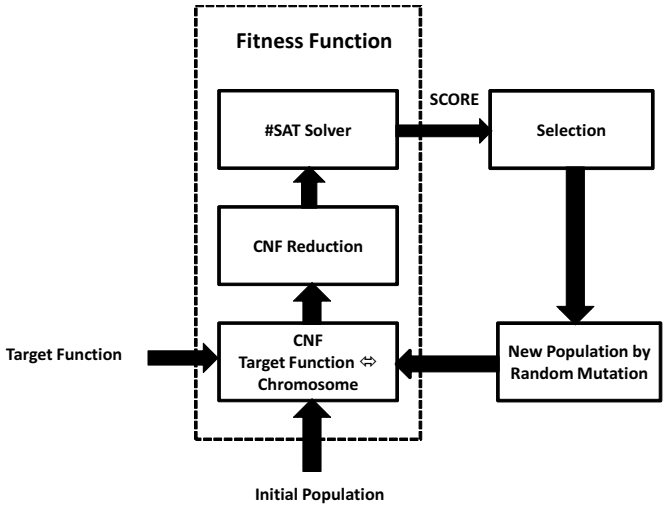


Fig. 4: Proposed Methodology.

The first step is to transform the boolean expression of the target function to its corresponding CNF, which is used in every iteration of the the digital circuit evolution. The next step is to form a combined CNF of the XOR operation corresponding to the boolean expression of chromosomes in the current population and the target function. The combined CNF is reduced as much as possible to minimize the computation overhead and is then given to the #SAT solver so that its fitness score can be calculated.

Chromosome with the best fitness score along with some of its mutated version form the new population for the next generation. This process iterates until a chromosome which is functionally equal to the target function is found or the maximum number of iterations is reached. We accept the target function and the initial population of chromosomes in the form of minimized minterms and maxterms, which

leads to a more compact and generic representation. If required, other propositional logic formulas representations, such as truth tables or netlists, may also be used.

Algorithm 2 provides the details of the implementation corresponding to the proposed methodology outlined above. The algorithm accepts the target function and its number of rows n_r and columns n_c, the seed value for the chromosomes and the digital circuit evolution parameters λ, ℓ and μ. The body of Algorithm 2 is primarily composed of the functions: RANDOM_CHR, FITNESS_FUNCTION and MUTATE. RANDOM_CHR returns a randomly generated chromosome using parameters ℓ, n_c and n_r.

Algorithm 2 #SAT based Fitness Scoring for Digital Circuit Evolution

Inputs:
target_function: target function
n_r: number of rows
n_c: number of columns
seed: a seed value for chromosome
λ: population size
ℓ: level back
μ: mutation rate
Output:
best_chromosome: chromosome with the best fitness score

```
1: for i=1 to λ+1 do
2:     if seed=0 then
3:         chromosome_i ←RANDOM_CHR(seed,n_r, n_c, ℓ)
4:     else
5:         chromosome_i ←seed
6:     end if
7: end for
8: while target chromosome found or max iterations reached do
9:     Initialize best_score with zero
10:    for i=1 to λ+1 do
11:        score←FITNESS_FUNCTION(chromosome_i)
12:        if score≥best_score then
13:            index←i
14:            best_score←score
15:        end if
16:    end for
17:    best_chromosome← chromosome_index
18:    if best_score=max_score or max iteration reached then
19:        break loop
20:    end if
21:    chromosome_1 ←best_chromosome
22:    for i=2 to λ+1 do
23:        chromosome_i ←MUTATE(best_chromosome,μ, ℓ)
24:    end for
25: end while
26: Print report for best_chromosome
```

FITNESS_FUNCTION accepts a chromosome and returns its fitness score by using our proposed #SAT solving. Whereas, the function MUTATE accepts a chro-

mosome and returns its mutated version using parameters μ and ℓ. The proposed methodology algorithm returns the chromosome that is found to be closest to the target function in the given number of iterations. In the next section, we present the implementation details associated with the FITNESS_FUNCTION, i.e., the #SAT solver that returns the fitness score between the target function and a given chromosome.

5. Implementation of the #SAT Algorithm for Fitness Scoring

In this section, we describe the proposed #SAT based fitness scoring by explaining our CNF conversion algorithm. Moreover, we also provide an overview of the implementation details of the proposed #SAT solver here.

5.1. *Implementation Details of the CNF Conversion Algorithm*

The proposed CNF conversion algorithm, given in Algorithm 3, accepts the target function (T) and candidate circuits or chromosomes (C) as inputs and returns their combined CNF. Both the target function (T) and the chromosome (C) share the same set of inputs $X = \{x_1, x_2, ...x_M\}$. The target function has N outputs, i.e., $Y = \{y_1, y_2, ...y_N\}$. Similarly, C has N outputs, i.e., $F = \{f_1, f_2, ...f_N\}$. The set of variables corresponding to the outputs of the intermediate nodes of C is $Z = \{z_1, z_2, ...z_{n_r \times n_c}\}$.

The function ENCODE_CNF, given in Algorithm 3, encodes the CNF as a disjunction, i.e., logical OR $\vee$, of XOR's between the respective outputs from the target function and the chromosome, i.e., the CNF for an N-output circuit will be encoded as $E = (y_1 \oplus f_1) \vee (y_2 \oplus f_2) \vee (y_3 \oplus f_3) \vee .....(y_N \oplus f_N)$. This way, E will be false if and only if all the corresponding outputs from target function and chromosome are equal. The first step in the CNF conversion process is to express $y_i's$ and $f_i's$ in terms of $x_i's$ in order to obtain an expression of E in terms of inputs $x_i's$ only. The function RESOLVE_Y finds the CNF expression for E by representing the $y_i's$ in terms of their corresponding $x_i's$ and returns the resulting simplified CNF in a variable CNF_{input}.

The above-mentioned tep is done by using the proposition resolution rules given in Table 2, where $\vee$ represents logical disjunction, $\wedge$ represents logical conjunction, i.e., AND, and $\neg$ represents the logical negation operator. These rules allow us to transform any propositional logic formula, with sub-formulas A, B and D, in its corresponding CNF.

The outputs $y_i's$ remain the same throughout the evolution process since they represent the target function whereas the outputs $f_i's$ alter in every iteration of the evolution process. In order to minimize the computation cost, we obtain the CNF_{input} only once and reuse it along with the current $f_i's$ to find the net CNF expression for every iteration. The function RESOLVE_F accepts CNF_{input} and returns the net CNF expression by representing the $f_i's$ in terms of their corresponding $x_i's$.

Algorithm 3 CNF Conversion

Inputs:
target_function: representation of the target truth table
chromosome: a candidate circuit
Output:
CNF: combined CNF of target function and chromosome

```
1: E←ENCODE_CNF(N)
2: CNF_input ←RESOLVE_Y(E,target_function)
3: CNF←RESOLVE_F(CNF_input,chromosome)
4: function ENCODE_CNF(N)
5:     Initialize E as empty set
6:     for i=1 to N do
7:         E←E∨(y_i ⊕ f_i)
8:     end for
9:     return E
10: end function
11: function RESOLVE_Y(E,target_function)
12:     for all clauses of E do
13:         for all literals do
14:             resolve y_i's in terms of its corresponding x_i's
15:         end for
16:     end for
17:     return E
18: end function
19: function RESOLVE_F(CNF_input,chromosome)
20:     CNF← CNF_input
21:     for all clauses of CNF do
22:         for all literals do
23:             resolve f_i's in terms of its corresponding x_i's
24:         end for
25:     end for
26:     return CNF
27: end function
```

Table 2: Resolution Rules for Some Common Gates.

Clause	**Resulting Clause**
$(\text{and}(a,b) \vee d)$	$(a \vee d) \wedge (b \vee d)$
$(\text{nand}(a,b) \vee d)$	$(\neg a \vee \neg b \vee d)$
$(\text{and}(a,\neg b) \vee d)$	$(a \vee d) \wedge (\neg b \vee d)$
$(\text{nand}(a,\neg b) \vee d)$	$(\neg a \vee b \vee d)$
$(\text{or}(a,b) \vee d)$	$(a \vee b \vee d)$
$(\text{nor}(a,b) \vee d)$	$(\neg a \vee d) \wedge (\neg b \vee d)$
$(\text{xor}(a,b) \vee d)$	$(a \vee b \vee d) \wedge (\neg a \vee \neg b \vee d)$
$(\text{xnor}(a,b) \vee d)$	$(a \vee \neg b \vee d) \wedge (\neg a \vee b \vee d)$

5.2. *Unsatisfiable Assignment Counting #SAT Solver*

In the proposed methodology, we need a #SAT solver that allows us to calculate the number of assignments for which the given CNF is unsatisfiable. It is impor-

tant to note that we are interested only in the number of unsatisfying assignments without specifically knowing them. The proposed #SAT solver algorithm is given in Algorithm 4.

Algorithm 4 Proposed SAT Solver

Input:
CNF: combined CNF of target function and chromosome
Output:
fitness score: measure of closeness between target function and chromosome

```
1: function SAT_SOLVER(CNF)
2:     initialize score to zero
3:     for i=1 to a do
4:         weight←WEIGHT(Φ_i,1)
5:         overlap_weight←OVERLAP_WEIGHT(Φ_i,a)
6:         net_weight←weight-overlap_weight
7:         score←score+net_weight
8:     end for
9:     return score
10: end function
11: function OVERLAP_WEIGHT(CL,p)
12:     pn←1
13:     for i=a-1 to 1 do
14:         tmp_pn ← CL∨Φ_i
15:         if tmp_pn is a valid clause then
16:             sign_pn=-1
17:             pn←pn+1
18:             temp←pn-2
19:             for j=1 to temp and temp 0 do
20:                 tmp_pn ← tmp_j ∨ Φ_i
21:                 if tmp_pn is a valid clause then
22:                     sign_pn = −1 × sign_j
23:                     pn←pn+1
24:                 end if
25:             end for
26:         end if
27:     end for
28:     for i=1 to pn-1 do
29:         overlap_weight=overlap_weight+WEIGHT(tmp_i, sign_i)
30:     end for
31:     return overlap_weight
32: end function
33: function WEIGHT(Φ,sign)
34:     K←number of literals in Φ
35:     weight← 2^(m−k)
36:     return sign×weight
37: end function
```

A clause is termed as unsatisfiable if no assignment of its literals makes it true. Let k denote the number of literals appearing in a given CNF clause. Considering that the total number of inputs is m, which is greater than or equal to k, the upper bound on the total number of input variable assignments for which the given clause

is unsatisfiable is 2^{m-k}. The function WEIGHT, used on Line 4 of Algorithm 4, accepts a CNF clause and calculates the number of its unsatisfying assignments.

Each clause is unsatisfiable for a unique set Φ_i of input assignments i. For fitness scoring, we are interested to find the union of all $\Phi_i's$, i.e.,

$$\bigcup_{i=1}^{i=q} \Phi_i \tag{1}$$

Different clauses may share the same unsatisfiable input variable assignments and thus the above mentioned union must be treated like the problem of overlapping sets. In a similar way, the function OVERLAP_WEIGHT, used on Line 5 of Algorithm 4, calculates the union, given in the above equation, for any number of clauses. We have used this net number of unsatisfying assignments as a measure of closeness, i.e., fitness score, between the target function and the chromosome.

For illustration purposes, we present the whole fitness calculation process of a simple single gate circuit, composed of an AND gate, in Table 3.

Table 3: An Example of the Fitness Calculation.

Target Function	$f1 = x1 \wedge x2$
Candidate Circuit	x1∨ x2
Encoded CNF	$(y1 \vee f1) \wedge (\neg y1 \vee \neg f1)$
CNF$_{input}$	$(x1 \vee f1) \wedge (x2 \vee f1) \wedge (\neg x1 \vee \neg x2 \vee \neg f1)$
CNF	$(x1 \vee x2) \wedge (x1 \vee x2) \wedge (\neg x1 \vee \neg x2) \wedge (\neg x1 \vee \neg x2)$
CNF after reduction	$(x1 \vee x2) \wedge (\neg x1 \vee \neg x2)$
Fitness Score	2

6. Experimental Results

We used the proposed method, described in the last two sections, to evolve circuits that exhibit the behaviour of adders, multipliers, multiplexers, even parity circuits, encoders, and a couple of LGSynth91 benchmarks. Moreover, we evolved the same circuits using state-of-the-art parallel simulation based CGP technique as discussed in Section 2.1, in order to compare the chromosome evaluation time of the two methods.

It is important to note that the simulation timing is based on running the analysis for all possibilities of the inputs unless the candidate circuit matches the specification before all combinations are tried out. Thus, in both of our experiments, there are no chances of getting a wrong circuit at the end.

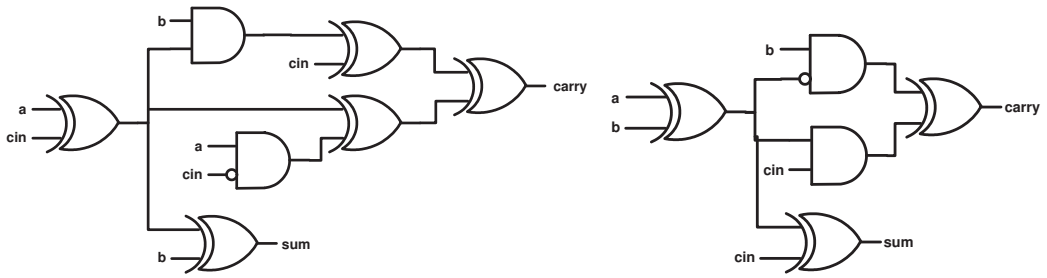
The experimental results are obtained by implementing the proposed methodology using Visual C++ ver. 6 and running on an Intel Duo core 1.86 Ghz processor based machine using the following parameters: i) The digital circuit evolution process has access to a subset of all possible functions, i.e., Γ=6,7,10,12, described in

Table 1, to simplify the evolution process. ii) A mutation rate $\mu = 4\%$ is chosen as it is known to give better convergence, iii) The level back variable ℓ is chosen to be equal to n_c in order to keep the search space larger.

We first present some of the evolved digital circuits using the proposed methodology. These circuits have been chosen because they highlight some unique characteristics of the proposed approach. Finally, we present a comparison table summarizing the mean evaluation times per chromosome for all the above mentioned circuits using conventional CGP and the proposed SAT-Based CGP for comparison purposes along with some discussions.

6.1. *Evolving the Full Adder Circuit*

We evolved the full adder circuit using the conventional CGP and the proposed SAT solving based CGP and the evolved circuits are given in Fig.5. Both circuits are structurally different, but functionality wise they behave as full adder circuits. Interestingly, both the circuits are different from the full adder circuit that is usually found in the text books or are designed using the conventional design rules. The conventional simulation based CGP technique took 998 generations and a mean evaluation time of 0.102 milliseconds per chromosome while the proposed SAT solving based technique took 168 generations and a mean evaluation time of 0.035 milliseconds per chromosome. This difference clearly demonstrates the effectiveness of the proposed SAT based scoring mechanism.



(a) (b)

Fig. 5: Full Adder (a) Conventional CGP (b) Proposed CGP.

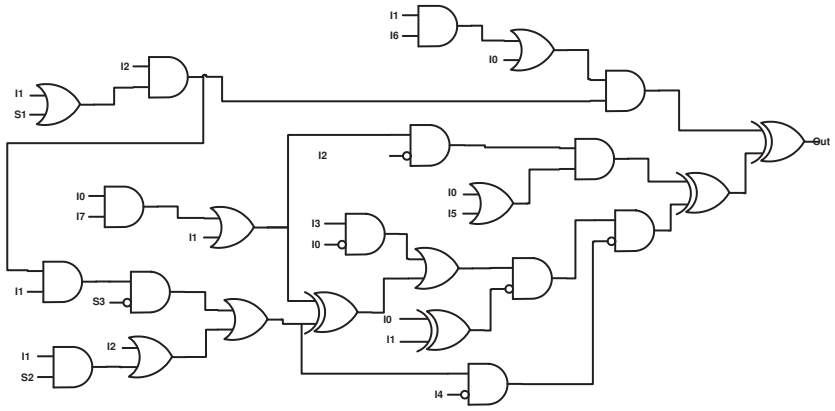
6.2. *Evolving the 8x1 Multiplexer Circuit*

The circuits of 8 line Mux evolved using the conventional simulation and the proposed SAT solving based CGP techniques are given in Fig.6.

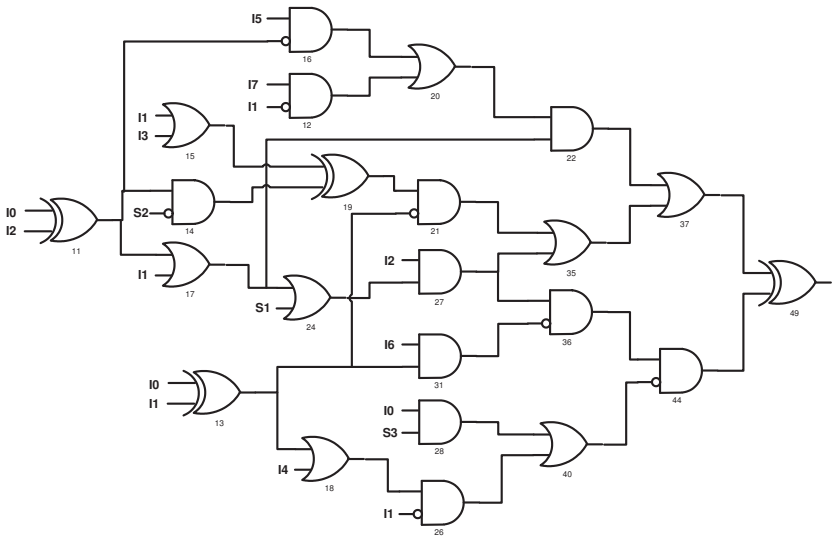
The large number of inputs for a 8x1 Mux makes it susceptible to the heavy computation requirement in the case of conventional simulation based CGP. Both of these circuits have different structures but have 23 2-input gates in total. On the other hand, one of the conventional 8x1 Mux implementations consists of 8 3-input

AND gates, 3 inverters and 1 8-input OR gate.

In terms of computation time, the conventional simulation based CGP took about 850k generations and a mean evaluation time of 1.06 milliseconds per chromosome. The proposed SAT solving based technique took about 100k generations and a mean evaluation time of 0.41 milliseconds per chromosome. This difference is mainly due to the effectiveness of the proposed SAT solving based scoring method for circuits, for which the number of CNF clauses have a linear relationship with the number of inputs.



(a)



(b)

Fig. 6: 8x1 Mux (a) Conventional CGP (b) Proposed CGP

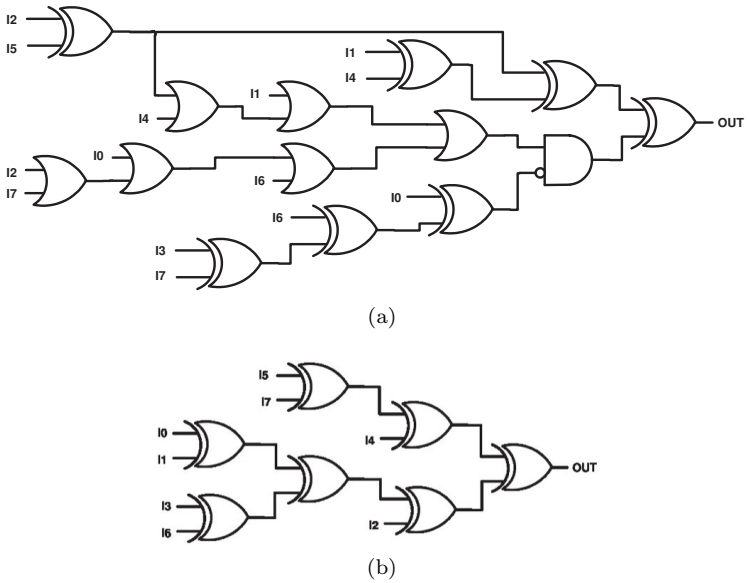


Fig. 7: 8-Bit Parity Circuit (a) Conventional CGP (b) Proposed CGP.

6.3. *Evolving the 8-Bit Parity Circuit*

The output of an 8 bit parity circuit is true if the number of ones in its input vector is odd. The CNF representation of parity circuits is known to be large and its size grows exponentially with the increase in the number of inputs. The evolved circuits of 8-bit parity circuits, using the conventional simulation, and the proposed SAT solving based CGP techniques are given in Fig.7 and are composed of 14 and 7 2-input gates, respectively. One of the most commonly used conventional implementations of a 8-bit parity circuit is also composed of 7 2-input XOR gates.

Thus, the proposed SAT solving based CGP technique generated an implementation for the 8-bit parity circuit that is more area-efficient than the simulation based CGP and is closer to the conventional implementation. In this case, the conventional simulation based CGP took about 36k generations and a mean evaluation time of 0.32 milliseconds per chromosome while the proposed SAT solving based technique took about 2k generations and a mean evaluation time of 0.7 milliseconds per chromosome.

The proposed SAT solving based CGP took more time in the fitness calculation process than the conventional simulation based CGP technique. This mainly happened because of the larger CNF representation of the circuit, which is composed of 256 clauses.

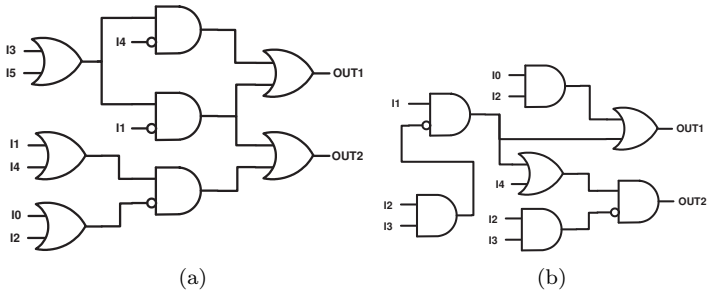


Fig. 8: c17 Circuit (a) Conventional CGP (b) Proposed CGP.

6.4. *Evolving the LGSynth91 c17 Benchmark*

The c17 is a LGSynth91 benchmark circuit with 5 inputs and 2 outputs [19]. The c17 circuits evolved, using conventional simulation based CGP and our proposed SAT solving based CGP techniques, are given in Fig.8 and both implementations consist of 8 2-input gates in different configurations. On the other hand this benchmark circuit is traditionally modelled using 6 2-input gates. The conventional simulation based CGP took about 29k generations and a mean evaluation time of 0.12 milliseconds per chromosome, while the proposed SAT solving based technique took about 66k generations and an evaluation time of 0.009 milliseconds per chromosome. We observed the maximum time gain for the c17 circuit and this is mainly due to the fact that it has a relatively low number (12) of CNF clauses in its CNF representation.

6.5. *Discussions*

We also evolved some other circuits and the results are summarized in Table 4.

The primary goal of presenting the above case studies was two-fold. Firstly, we wanted to illustrate the correctness of the proposed technique, i.e., it can evolve all kinds of circuits with the desired functionality. All the four examples demonstrate this point. The second purpose was to illustrate the effectiveness of the proposed technique in terms of fitness calculation time for cases where the number of CNF clauses does not grow exponentially with the number of inputs.

We chose all the major circuits that have been evolved using various digital circuit evolution techniques and the proposed technique was able to successfully evolve all of them. In terms of evolution time, it can be observed that the fitness calculation time is lower for most cases, depicted by the highlighted ones are the rest, compared to the traditional CGP with parallel simulation, which is known to have the fastest evolution time so far [21]. The circuits for which the proposed technique did not perform well, depicted by the highlighted cases, are the ones in which the number of CNF clauses grows exponentially with the number of inputs.

Table 4: Mean Evaluation Time Per Chromosome for Conventional CGP t_{cgp} and SAT-Based CGP t_{mcgp}.

Circuit	Number of Inputs	Number of Outputs	Minimized input CNF Clauses	n_r x n_c	Mean Time per Chromosome		Time Gain
					t_{cgp} ms	t_{mcgp} ms	%
Full Adder	3	2	14	1x10	0.102	0.035	291.43
3 Bit Adder	7	4	48	1x25	0.21	0.53	39.62
2x2 Multiplier	4	4	22	1x10	0.12	0.035	342.86
3x3 Multiplier	6	6	102	1x40	0.15	2.2	6.82
Parity 5	5	1	32	1x12	0.115	0.018	638.89
Parity 6	6	1	64	1x15	0.15	0.22	68.18
Parity 8	8	1	256	1x20	0.32	0.7	45.71
4x2 Encoder	4	2	8	1x15	0.095	0.018	527.78
8x2 Encoder	8	2	24	1x20	0.35	0.06	583.33
2x1 Mux	3	1	4	1x10	0.11	0.017	647.06
4x1 Mux	6	1	8	1x15	0.17	0.072	236.11
8x1 Mux	11	1	16	1x30	1.06	0.41	258.54
Lgsynth91 c17	5	2	12	1x13	0.12	0.009	1333.33
Lgsynth91 majority	5	1	11	1x15	0.094	0.016	587.50
16 line MUX and 16 Decoder 10000 chromosome run							
16x1 MUX	20	1	32	1x40	536	2.5	21440
16x4 Encoder	16	4	48	1x40	44.8	5.2	861.54

7. Conclusions

This paper presents a novel digital circuit evolution approach based on the principles of SAT solving. The main idea is to leverage upon the computational efficiency of #SAT solvers to expedite the process of fitness scoring, which is traditionally done using exhaustive simulations. We proposed a CGP based digital circuit evolution technique, where the equivalence problem of the target function and the given chromosome is represented in terms of a CNF and then the number of input assignments, for which this CNF is unsatisfiable, can be found using #SAT solving algorithms. This number is used to judge the fitness of the chromosomes and the digital circuit evolution is carried out using the traditional genetic programming based approach.

We implemented the proposed methodology in C++. The experimental results illustrate the effectiveness of the proposed approach as we are able to correctly evolve all the state-of-the-art evolved digital circuits. Moreover, using the proposed approach, significant reduction in evolution time was observed for circuits in which the number of CNF clauses is a linear function of the number of inputs.

To the best of our knowledge, #SAT-based method has never been used in the

domain of digital circuit evolution to measure the degree of closeness between two circuits. Thus, this paper opens the doors to a new area of research in both of these domains. The proposed idea can be used in conjunction with the post-evolution reduction techniques and other already proposed efficient CGP techniques, such as 40, to evolve more complex digital circuits, in terms of gate count and inputs with less computational resources. Moreover, smarter fitness scoring criterion than the ones reported, in this paper, can be explored by the SAT solving community to further reduce the computational times and optimize the evolved circuits. A worth investigation direction in this regard would be to use SMT solvers [35] instead of SAT solvers for fitness scoring. Moreover, crossover operations can also be used besides random mutations to optimize the results of the proposed methodology.

References

1. Minisat, http://minisat.se, Accessed in July 2017.
2. F.V. Andrade, L.M. Silva, and A.O. Fernandes. Improving SAT based combinational equivalence checking through circuit preprocessing. *Computer Design*, (2008), pp. 40–45.
3. A. Biere. PicoSAT Essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, **2**, 201, (2008).
4. J. Cong and K. Minkovich. Optimality study of logic synthesis for LUT-based FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **26** 230, (2007).
5. S. Disch and C. Scholl. Combinational equivalence checking using incremental SAT solving, output ordering, and resets. *Design Automation Conference, Asia and South Pacific*, (2007), pp. 938–943.
6. Z. Gajda and L. Sekanina. When does cartesian genetic programming minimize the phenotype size implicitly? *Genetic and evolutionary computation*, (2010) pp. 983–984.
7. K. Glette, J. Torresen, and M. Yasunaga. An online EHW pattern recognition system applied to face image recognition. *Applications of Evolutionary Computing*, (2007), pp. 271–280.
8. D.E Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. *Foundations of Genetic Algorithms*, (1991), pp. 69–93.
9. T.G.W Gordon and P.J. Bentley. Towards development in evolvable hardware. *Evolvable Hardware*, (2002), pp. 241–250.
10. G. Greenwood and A.M. Tyrrell. *Introduction to Evolvable Hardware, 1st edn.*. (Wiley-IEEE, 2007).
11. P.C. Haddow, G. Tufte, and P. Remortel. Shrinking the genotype: L-systems for ehw? *Evolvable Systems: From Biology to Hardware*, (2001) pp. 128–139..
12. S.L. Harding, J.F. Miller, and W. Banzhaf. Self-modifying cartesian genetic programming. *Cartesian Genetic Programming*, (2011), pp. 101–124.
13. K. Imamura, J.A. Foster, and A.W. Krings. The test vector problem and limitations to evolving digital circuits. *Evolvable Hardware*, (2000), pp. 75–79.
14. P. Kaufmann and M. Platzner. Advanced techniques for the creation and propagation of modules in cartesian genetic programming. *Genetic and evolutionary computation*, (2008), pp. 1219–1226.
15. S. Kemper. Sat-based verification for timed component connectors. *Electronic Notes in Theoretical Computer Science*, **255**, 103, (2009).
16. J.R. Koza. *Genetic Programming, 1st edn.*. (MIT Press, 1992).

17. J.R. Koza, F.H. Bannett, D. Andre, and M.A. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving, 1st edn.*. (Morgan Kaufmann, 1999).
18. J.R. Koza, M.A Keane, M.J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence, 1st edn.*. (Kluwer, 2003).
19. LGsynth91 Benchmark, https://ddd.fit.cvut.cz/prj/Benchmarks/, Accessed in July 2017.
20. D. Mange, M. Sipper, A. Stauffer, and G Tempesti. Toward robust integrated circuits: The embryonics approach. *Proceedings of the IEEE*, (2000) pp. 516–541.
21. J.F. Miller, D. Job, and V. Vassilev. Principles in the evolutionary design of digital circuits part i. *Genetic Programming and Evolvable Machines*, **1**, 7, (2000).
22. J.F. Miller and S.L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, **10**, 167, (2006).
23. J.F. Miller and P. Thomson. Cartesian genetic programming. *Genetic Programming*, (2000), pp. 121–132.
24. J.F. Miller, P. Thomson and T. Fogarty. Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study, *Genetic Algorithms and Evolution Stratagies in Engineering and Computer Science*, (1997), pp. 105-131.
25. M. Murakawa, S. Yoshizawa, I. Kajitani, T. Furuya, M. Iwata, and T. Higuchi. Hardware evolution at function level. *Parallel Problem Solving from Nature*, (1996), pp. 62–71.
26. E.I. Prez, C.C. Coello, and A.H. Aguirre. Extracting and re-using design patterns from genetic algorithms using case-based reasoning. *Engg. Optimization*, **35**,121, (2003).
27. A. Sagahyroon, F.A. Aloul, and A. Sudnitson. Using sat-based techniques in low power state assignment. *Journal of Circuits, Systems and Computers*, **20**, 1605, (2011).
28. L. Sekanina. *Evolvable Components: From Theory to Hardware Implementations, 1st edn.*. (Springer, 2004).
29. L. Sekanina. Evolutionary design of digital circuits: Where are current limits? *Adaptive Hardware and Systems*, (2006), pp. 171–178.
30. L. Sekanina. Evolvable hardware: From applications to implications for the theory of computation. *Unconventional Computation*, (2009), pp. 24–36.
31. A. P. Shanthi and R. Parthasarathi. Practical and scalable evolution of digital circuits. *Appl. Soft Comput.*, **9(**, 618 (2009).
32. S.Z. Shazli and M.B. Tahoori. Soft error rate computation in early design stages using boolean satisfiability. *ACM Great Lakes Symposium on VLSI'09*, (2009), pp. 101–104.
33. E. Stomeo, T. Kalganova, and C. Lambert. Generalized disjunction decomposition for evolvable hardware. *EEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics, I*, **36**, 1024 (2006).
34. M. Thurley. sharpsat counting models with advanced component caching and implicit bcp. *Theory and Applications of Satisfiability Testing - SAT 2006*, (2006), pp. 424–429.
35. L. Tianhai, M. Nagel, and M. Taghdiri. Bounded program verification using an smt solver: A case study. *Software Testing, Verification and Validation*, (2012), pp. 101–110.
36. J. Torresen. A divide-and-conquer approach to evolvable hardware. *Evolvable Systems: From Biology to Hardware*, (1998), pp. 57–65.
37. J. Torresen. A scalable approach to evolvable hardware. *Genetic Programming and Evolvable Machines*, **3**, 259, (2002).
38. G.S. Tseitin. On the complexity of derivation in propositional calculus. *Automation of Reasoning*, (1983), pp. 466–483.
39. Z. Vasicek and L. Sekanina. Formal verification of candidate solutions for post-synthesis evolutionary optimization in evolvable hardware. *Genetic Programming and Evolvable Machines*, **12**, 305, (2011).

40. Z. Vasicek and L. Sekanina. How to evolve complex combinational circuits from scratch? *Evolvable Systems*, (2014), pp. 133–140.
41. Z. Vašíček, M. Žádník, L. Sekanina, and J.Tobola. On evolutionary synthesis of linear transforms in FPGA. *Evolvable Systems: From Biology to Hardware*, (2008), pp. 141–152.
42. V.K. Vassilev and J.F. Miller. Scalability problems of digital circuit evolution evolvability and efficient designs. *Evolvable Hardware*, (2000), pp. 55–64.
43. M.N. Velev. Efficient translation of boolean formulas to CNF in formal verification of microprocessors. *Design Automation Conference, Asia and South Pacific*, (2004), pp. 310–315.
44. J.A. Walker and J.F. Miller. The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, **12**, 397, (2008).
45. S. Zhan, J.F. Miller, and A.M. Tyrrell. A developmental gene regulation network for constructing electronic circuits. *Evolvable Systems: From Biology to Hardware*, (2008), pp. 177–188.