INTERNATIONAL JOURNAL OF ENGINEERING AND MANAGEMENT SCIENCES
© 2004 -16 Society For Science and Nature (SFSN). All Rights Reserved
www.scienceandnature.org

GENERATING TEST PATTERNS USING EVOLUTIONARY ALGORITHM IN ARTIFICIAL INTELLIGENCE

[1]Sangeeta [2]Harsharandip S.Kler
[1]Department of CSE, DAVIET Jalandhar, Punjab, India
[2]Department of ECE, Govt. ITI, Jargon, Punjab, India

ABSTRACT
This paper presents a brief introduction to artificial Intelligence and FPGAs [8][11]. We have discussed that how test patterns for stuck at faults can be formulated in terms of CNF form [4] and this CNF form can be used to generate test patterns using multiobjective genetic algorithm. We have proposed that by applying a multi-objective genetic algorithm on this CNF form we can increase number of instances to satisfy boolean equation.

KEYWORDS: Field Programmable Gate Array (FPGAs), Conjunctive Normal Form (CNF), Multi-objective Genetic Algorithm (MOGA), Artificial Intelligence (AI)

INTRODUCTION

Artificial Intelligence

Artificial Intelligence (AI) is a branch of Science which deals with helping machines find solutions to complex problems in a more human-like fashion. This generally involves borrowing characteristics from human intelligence, and applying them as algorithms in a computer friendly way. A more or less flexible or efficient approach can be taken depending on the requirements established, which influences how artificial the intelligent behavior appears[13] From the perspective of intelligence artificial intelligence is making machines "intelligent" -- acting as we would expect people to act. Intelligence requires knowledge of Expert problem solving - restricting domain to allow including significant relevant knowledge optimization takes places before decision making.

From a programming perspective, AI includes the study of symbolic programming, problem solving, and search. Typically AI programs focus on symbols rather than numeric processing. Artificial Intelligence AI programming languages include: – LISP, developed in the 1950s, is the early programming language strongly associated with AI. LISP is a functional programming language with procedural extensions. LISP (LISt Processor) was specifically designed for processing heterogeneous lists -- typically a list of symbols. Features of LISP are run- time type checking, higher order functions (functions that have other functions as parameters), automatic memory management (garbage collection) and an interactive environment. – The second language strongly associated with AI is PROLOG. PROLOG was developed in the 1970s. PROLOG is based on first order logic. PROLOG is declarative in nature and has facilities for explicitly limiting the search space. – Object-oriented languages are a class of languages more recently used for AI programming.

Many problems in AI can be solved in theory by intelligently searching through many possible solutions [14] .Reasoning can be reduced to performing a search. For example, logical proof can be viewed as searching for a path that leads from premises to conclusions, where each step is the application of an inference rule.[14] Planning algorithms search through trees of goals and sub goals, attempting to find a path to a target goal, a process called means-ends analysis.[13] Robotics algorithms for moving limbs and grasping objects use local searches in configuration space.[13] Many learning algorithms use search algorithms based on optimization.

Simple exhaustive searches [13] are rarely sufficient for most real world problems, the search space (the number of places to search) quickly grows to astronomical numbers. The result is a search that is too slow or never completes. The solution, for many problems, is to use "heuristics" or "rules of thumb" that eliminate choices that are unlikely to lead to the goal (called "pruning the search tree"). Heuristics supply the program with a "best guess" for the path on which the solution lies. [13] Heuristics limit the search for solutions into a smaller sample size. [14]

A very different kind of search came to prominence in the 1990s, based on the mathematical theory of optimization. For many problems, it is possible to begin the search with some form of a guess and then refine the guess incrementally until no more refinements can be made. These algorithms can be visualized as blind hill climbing. We begin the search at a random point on the landscape, and then, by jumps or steps, we keep moving our guess uphill, until we reach the top. Other optimization algorithms are simulated annealing, beam search and random optimization. [15]

ARTIFICIAL INTELLIGENCE AND EVOLUTIONARY ALGORITHM

Evolutionary computation uses a form of optimization search. For example, they may begin with a population of organisms (the guesses) and then allow them to mutate and recombine, selecting only the fittest to survive each generation (refining the guesses). Forms of evolutionary computation include swarm intelligence algorithms (such as ant colony or particle swarm optimization) [15] and evolutionary algorithms (such as genetic algorithms, gene expression programming, and genetic programming).[10]

Implementation of Evolutionary Algorithm

1. Generate the initial population of individuals randomly (first generation)
2. Evaluate the fitness of each individual in that population.
3. Repeat on this generation until termination (time limit, sufficient fitness achieved, etc.)
    a) Select the best-fit individuals for reproduction - (parents)
    b) Breed new individuals
    c) Evaluate the individual fitness of new individuals.
    d) Replace least-fit population with new individuals.

Artificial Intelligence Used In Combinational Circuit Testing

During recent years, great effort is put to overcome test generation complexity problem. Artificial intelligence methods are therefore gained much of attention. One among these techniques is evolutionary algorithms or often referred as genetic algorithms. Earlier genetic approach for combinational circuits was represented in [14]. The key feature there was the method of monitoring circuit activity network. The simplicity, robustness, efficiency and effectiveness of GA make them a promising tool for complex applications. GA maintains a population pool of candidate solutions called strings or chromosomes. Each string is associated with a fitness value determined by a user defined fitness function.GA starts with an initial population typically generated randomly and the evolutionary process of reproduction, crossover and mutation are used to generate an entirely new population from the existing population. The new population and the existing population compete for membership in the generation's membership pool. Selection of the chromosomes for new population is governed by the replacement strategy.

The old population is discarded. The sequence of selection, crossover, and mutation completes one generation cycle.GA progresses through generations until the goal is reached such as fixed number of generations. So GA algorithm can be used to optimize the process of exploring.

Automatic Test Pattern Generation Using SAT

- Given a Boolean formula F (x_1, x_2, x_n), Boolean satisfiability (SAT) asks if there is an assignment to the variables, x_1, x_2, x_n, such that F evaluates to 1. If such an assignment exists, F is said to be *satisfiable*, otherwise, it is *unsatisfiable*. A SAT solver serves to answer the Boolean satisfiability problem.
- For practical purposes, modern day SAT solvers work on Boolean formulae in Conjunctive Normal Form (CNF). Boolean formulae in CNF consist of a conjunction of clauses. A clause is a disjunction (logical OR) of literals and a literal is any Boolean variable, *x belongs* {0,1}, or its complement. Any Boolean formula can be converted to CNF

$$(\overline{A}+B+C).(A+B+\overline{C})$$

A Boolean formula in CNF form[8]

- In CNF, the problem of SAT can be rephrased to the following: Given a Boolean formula, $F(x_1; x_2;\ldots; x_n)$, in Conjunctive Normal Form (CNF), seek an assignment to the variables, $x_1; x_2;\ldots;x_n$, such that each clause has at least one literal evaluating to 1.

Applying Multi Objective Genetic Algorithm to Test Pattern Generation

MOGA can be used in ATPG for exploring the work space. In genetic terms every test vector is considered as a chromosome and set of test vector is called as population. The ATPG algorithm performs in two phases. In the first phase the initial population is being generated with the help of pseudo random process. In the second phase the GA phase the test vectors are evolved based on fitness function[6].The fitness function used is

Fitness = NFi

Where NFi is the number of faults detected.

```
{
        FL= {total number of faults}
        initial pop=phase I (FL);
        if (FL =NULL)
        break;
        phase II (initial pop, FL);
}
```

Figure 4.1: Pseudo-code of overall GA based test pattern generation

Phase I

In this phase the initial sequences composed of M vectors are generated based on pseudo random process. The generated sequences are fault simulated for the faults in the fault list. If the sequence detects fault that fault is removed from the fault list and the corresponding sequence is added into the solution set. If no faults are detected by the sequence, then the last sequence generated in the corresponding cycle is added to the set. This process is repeated for max_iter.

Function Phase I

```
        initial pop (FL)
        for (i=0; i<max_iter, i++)
        {
            initial pop=phase I(FL);
            randomly generate sequences of length L;
                for (each sequence)
{
    if sequence detects faults in the fault list
    {
        add sequence to the test set;
        drop the faults detected by that sequence;
    }
}
return (initial population);
}
```

Figure 4.2: The Pseudo-code of Phase I

Phase II

The initial population of GA is composed of the sequences generated in phase I. To generate a new population from the existing one, two individuals (parents) are selected and crossed to create two entirely individuals (child) and each child is mutated with some small mutation probability. The selection operator is rank based selection. In rank based selection, the solutions are sorted according to their fitness from the worst (rank 1) to the best (rank N).Each member in the sorted list is assigned a fitness equal to the rank of the solution in the list. Thereafter the proportionate selection operator is applied with the ranked fitness value and better solutions are chosen. The two parents are crossed to create two entirely new individuals (i.e.) child and each child is mutated with some small mutation probability. The two new individuals are than placed in the new population and the process continues until the generation is entirely filled. The previous population is discarded. Crossover used is one point crossover. A crossover probability of 1 and mutation probability of 0.01 is used in all circuits. The no_gen is assumed to be 8, to reduce the execution time. During test generation pop_size of 16 is used.

```
Function Phase II
{
    Initial pop from phase1;
    for (l=0;l<no_gen;l++)
    {   for (k=0; k<popsize;k++)
        {   select two individuals from
            population;
            apply crossover with probability 1;
            apply mutation with probability 0.01;}
            compute fitness of the individuals;
            for (each sequence)
            if (sequence detects the faults in the fault list
            { add sequence to the solution set;
                drop the faults detected by the
                sequence;
            }
        }
    }
```

Figure 4.3: Pseudo-code of Phase II

RESULTS

All experiments have been performed using the MATLAB. In all cases the population size is fixed, the standard 1-point and 2-point crossover operator has been applied at a 50% rate, the mutation is 0.1% and selection is performed using the roulette wheel selection algorithm. The algorithm scans the genes in a random order each gene is flipped, and the flipped is accepted if the fitness score is maximum. When all the genes have been tested the next generation is formed from the previous generation.

Having formulated FPGA testing problem, as SAT instances as an optimization problem using the Multiobjective Genetic Algorithm, there are some interesting issues concerning convergence to solution. First of all, whenever a candidate solution evaluates to 1, we know that the solution has been found and search can be terminated. Conversely, there is strong motivation to continue the search until a solution is found.

The difficulty, of course is that on any particular run there is no guarantee that a solution will be found in a reasonable amount of time due to the increasing homogeneity of the population as the search proceeds. Parameters for measuring the quality, computational cost and time of GA.

In order to obtain statistically significant results, several runs are required for different parameters on the SAT instances. The quality achieved is measured by the fault detection ratio which represents the ratio of number of clauses satisfied divided by total number of clauses. Second parameter, which is used to measure the computational cost of the genetic algorithm, is the time to evaluate the output. This measure does not depend upon the machine used and on the actual implementation. We specify a search space of size 2^n and total number of iterations taken to find the test vector

Tables & Graphs showing the Results

Table 1, 2 & 3 give the complete testing solution with varying number of variables from 8,16,32,64,128 respectively. The number of I/P is given by n and total number of possible combinations can be 2^n.So if value of n is 3 there can be possible 8 combinations for a circuit. The number of variables and clauses are sufficient to explain the order of magnitude of the FPGA testing problem. All simulation is being made by varying the number of iterations. Number of variables are fixed as input. Two parameters are observed for different number of iterations.

First is Fault Coverage ratio and second is time.

The columns in the table record the following data: the number of Boolean variables correspond to the Testing function, number of clauses in the Testing function, Fault Coverage of the algorithm. By Changing the number of generations and keeping other parameters same

Table 1: Performance of FPGA Testing using Multi-objective GA with 1 Generations

S.No	No of variables	No. of Clauses Satisfied	Time	Fault coverage
1	8	1594	0.06	79.6
2	16	2452	0.04	70.0
3	32	4636	0.01	87.4
4	64	9204	0.09	92.0
5	128	14478	0.32	91.0

In order to find out the any relation between these parameters experiment is being carried out with varying number of generations. As shown in [see table 5.1] the variables are being varied for fixed generation i.e. 1 in this case. For 8 variables we have total 20 number of clauses. Fault coverage ratio is calculated for 100 number of runs. It shows the total number of clauses satisfied are 1578 and total number of clauses specifically for this case are 2000.

Table 2: Performance of FPGA Testing using Multi-objective GA with 10 Generations

S.No	No of variables	No. of Clauses Satisfied	Time	Fault coverage
1	8	1588	0.03	79.4
2	16	2439	0.06	69.6
3	32	4608	0.04	86.9
4	64	9167	0.06	91.6
5	128	14465	0.37	90.9

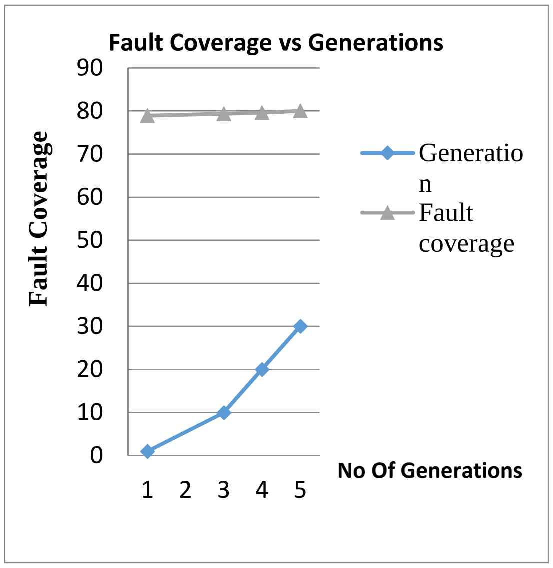
For next case number of generations are being increased to 10.The number of variables are again varied and number of runs are fixed as 100.In this case fault coverage ratio improves as compared to the previous case [see table 5.2] for each variable. By changing the number of generations to 20 keeping other parameters same now generations are increased with gap of 10 to get the relation between fault coverage and generations .Other parameters like number of runs, number of variables are kept same. Every time number of generations are increased fault coverage ratio improves. Number of satisfied clauses are increased from 1588 to 1594.The time required for this calculation is 0.06 ms.

Table3: Performance of FPGA Testing using Multi-objective GA with 20 Generations

S.No	No of variables	Clauses Satisfied	Time	Fault coverage
1	8	1578	0.07	78.9
2	16	2369	0.1	67.6
3	32	4600	0.10	86.7
4	64	9129	0.140	91.2
5	128	14414	0.37	90.6

In order to analyze the results, graphs are being constructed.

Figure 1: Fault coverage vs. Generations



CONCLUSION

Graph shows the effect of increasing the number of generations with fault coverage ratio. Each generation takes a different value as input so every generation produces different value for fault coverage ratio. Fault coverage ratio increases linearly with number of generations.
From tables [1-3] and their graph it has been shown that Multiobjective Genetic Algorithm is applied to variety of SAT instances of FPGA test pattern generation problem, which gives competitive results. It has been concluded from the experiments that as the problem size grows up i.e. number of variables and clauses are increased in the fault coverage ratio increases.

REFERENCES

[1] Carlos M. Fonsecay and Peter J. Flemingz "An Overview of Evolutionary Algorithms in MultiobjectiveOptimization", April 7, 1995

[2] Paolo Prinetto, Maurizio Rebaudengo, and Matteo Soriza "GATTO: A Genetic Algorithm for Automatic Test Pattern Generation for Large Synchronous Sequential Circuits" IEEE TRANSACTIOIVS ON COMPUTER-AIDED DESIGN ,VOL. 15, NO. 8, AUGUST 1996

[3] V. Rajesh, Ajai Jain "Automatic Test Pattern Generation for Sequential Circuits Using Genetic Algorithms" 1063-9667/9 IEEE 1997

[4] Yong Chang Kim and Kewal K. Saluja" Sequential test generators: past, present and future "Integration,the VLSI Journal Volume 26,Issues 1-2,1 ,Pages 41-54 December 1998

[5] Carlos A Coello Coello., "A Comparative survey of Evolutionary based Multiobjective Optimization" *December 1998*

[6] Li Shen "Genetic Algorithm Based Test Generation for Sequential Circuits" Institute of Computing Technology, Beijing May 2000

[7] Ying Gao Lei Shi Pingjin Yao "Study on Multi-Objective Genetic Algorithm" July,2000

[8] Arslan, T. Horrocks, D.H. Ozdemir, E. Sch. of Eng., Univ. of Wales Coll. of Cardiff "Structural synthesis of cell-based VLSI circuits using a multi-objective genetic algorithm " Electronics Letter Volume: 32 Issue 7 ,651 - 652 ISSN: 0013-5194 ,06 August 2002

[9] Gregor Papa ,Tomasz Garbolino ∗,Franc Novak ,Andrzej H lawiczka Deterministic Test Pattern Generator Design With Genetic Algorithm Approach Journal of ELECTRICAL ENGINEERING,VOL.58,NO.3,121–127,2007

[10] Charles Stroud, John Sunwoo, Srinivas Garimella, and Jonathan Harris Built-In Self-Test for System-on-Chip: A Case Study0-7803-8580-2/copyright IEEE 2004

[11] Michael S. Hsiao Virginia Tech, Blacksburg, Virginia VLSI Principles and Architecture, Pages 161-262 2006

[12] S.Jayanthy M.C.Bhuvaneswari Sri Ramakrishna Engineering College, Coimbatore, India P.S.G. College of Technology, Coimbatore, India "Simulation Based ATPG for Crosstalk Delay Faults in VLSI circuits using Genetic Algorithm ICGST- AIML journal, ISSN: 1687-4846, Volume 9, Issue 2, December 2007

[13] McCarthy, John *(12 November 2007).* "What Is Artificial Intelligence"

[14] *Rajani, Sandeep (2011).* "Artificial Intelligence – Man or Machine" *(PDF). International Journal of Information Technology and Knowledge Management* ***4*** *(1): 173–176.*

[15] L. Padma Suresh, Subhransu Sekhar Dash, Artificial Intelligence and Evolutionary Algorithms in Engineering Systems: Proceedings of ICAEES 2014, Volume 1