Exploration of Sequential Depth by Evolutionary Algorithms

Nicole Drechsler Rolf Drechsler

Institute of Computer Science
University of Bremen
28359 Bremen, Germany
{nd,rd}@informatik.uni-bremen.de

Abstract

Verification has become one of the major bottlenecks in today's circuit and system design. Up to 80% of the overall design costs are due to checking the correctness. Formal verification based on Bounded Model Checking (BMC) is a very powerful method that allows to prove the correctness of a device. In BMC the circuits behavior is considered over a finite time interval, but for the user it is often difficult to determine this interval for a given Device Under Verification (DUV).

In this paper we present a simulation based approach to automatically determine the sequential depth of a Finite State Machine (FSM) corresponding to the DUV. An Evolutionary Algorithm (EA) is applied to get high quality results. Experiments are given to demonstrate the efficiency of the approach.

1. Introduction

Modern circuits contain up to several hundred million transistors. In the meantime it has been observed that verification becomes the major bottleneck in circuit and system design, i.e. up to 80% of the overall design costs are due to verification. This is one of the reasons why recently several methods have been proposed as alternatives to classical simulation, since it cannot guarantee sufficient coverage of the design. E.g. in [2] it has been reported that for the verification of the Pentium IV more than 200 billion cycles have been simulated, but this only corresponds to 2 CPU minutes, if the chip is run with 1 GHz.

As alternatives, formal verification or symbolic simulation have been proposed and in the meantime these have been successfully applied in many industrial projects. To allow for an early detection of design errors, model checking has been used. While "classical" CTL-based model checking [4] can only be applied to medium sized designs, approaches based on *Bounded Model Checking* (BMC) as discussed in [3] give very good results when used for complete blocks with up to 100k gates.

But there is one inherent problem when applying BMC: The circuit is considered over a fixed time interval and to give complete proofs it is important to determine the sequential depth of the circuit. Recently in [16] an approach based on simulation in combination with a toggle-heuristic has been proposed, but experiments have shown that the method might result in over- or under-approximations and often gives sub-optimal results. This makes the technique hard to use for a designer or a verification engineer. An exact solution to this problem based on a problem formulation as quantified Boolean functions has been proposed in [13]. A SAT-solver is applied to compute the optimal result, but due to the complexity of real-world circuits this technique cannot be applied to larger problem instances.

In this paper we present a simulation based algorithm for computation of the sequential depth of FSMs. The quality of the simulated vectors is evaluated using techniques from *Evolutionary Algorithms* (EAs). It has been observed that EAs work very well in testing applications [6, 5, 14, 12, 10] and here the underlying problem is very similar. Experiments show that the same quality can be obtained as the exact approach but using simulation techniques only. By this, the EA technique combines the best of the two approaches from [16] and [13], i.e. we get the optimal results but for the evaluation no time consuming proof techniques, like BDD or SAT, are used, but only simulation that can be carried out in linear time in the circuit size.

The paper is structured as follows: First, basic definitions of sequential circuits and sequential depth computation are outlined. Then the proposed EA for depth approximation is presented. Experimental results show the quality of the presented approach and finally, the paper is summarized.

2. Preliminaries

A synchronous sequential circuit can be described using a *Finite State Machine* (FSM). An FSM is a 5-tuple $M = (I, O, S, \delta, \lambda)$, where I is the input set, O is the output set and S is the set of states. $\delta : I \times S \rightarrow S$ is the next-state function and $\lambda : I \times S \rightarrow O$ is the output function. Since we consider a gate level realization of the FSM, we have $I = \mathbf{B}^k$, $O = \mathbf{B}^l$, and $S = \mathbf{B}^m$ with

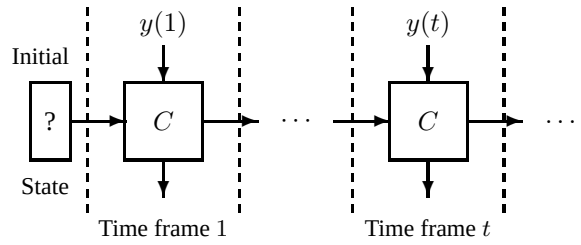

Figure 1. Iterative description of a sequential circuit

$\mathbf{B} = \{0, 1\}$. k denotes the number of primary inputs, l denotes the number of primary outputs, and m denotes the number of memory elements. The functions δ and λ are computed by a combinational circuit C. The inputs (outputs) of the combinational circuit, which are connected to the outputs (inputs) of the memory elements, are called secondary inputs (outputs). Sometimes the secondary inputs are called *present state variables* and the secondary outputs are called *next state variables*.

For the description of our algorithms we use the following notations: $X = x(1), \ldots, x(n)$ denotes the input sequence of depth n. s_i denotes the next state defined by $x(i)$ and s_{i-1}, $1 \leq i \leq n$.

Using these notations the next state is given by

$$s(s_0, t) \quad = \quad \begin{cases} s_0 & \text{if } t = 0 \\ \delta(x(t),\ s(s_0, t-1)) & \text{otherwise} \end{cases}$$

In doing so, we consider a synchronous sequential circuit as an iterative network (see Figure 1).

The state transition graph of an FSM is a labeled directed graph $T = (V, E)$ where each node $v \in V$ corresponds to a state s_i, $0 \leq i \leq |S| - 1$, of M, and each edge $e = (v, w)$, $v, w \in V$, corresponds to a transition from state s_i to state s_j. The edge is labeled with $y \in I^k$ which is the input vector that affects the transition from s_i to s_j, i.e. $\delta(y, s_i) = s_j$, $0 \leq i, j \leq |S| - 1$.

A *path* is a sequence of nodes v of T where all nodes are different. Using the definitions above, the *sequential depth* of an FSM is given as follows:

> Consider an FSM M and its corresponding state transition graph T with a single initial state s_0. Find a path of maximum length starting in s_0 such that each node along the path is visited only once and additionally, the path has maximum length.

Example 1 In Figure 2 a state transition graph with four states is illustrated. If the initial state is 00, only path 00-10-11-01 with length 3 exists. All other paths have a shorter length, i.e. paths 00-01-11 and 00-10-11 have length 2. The resulting sequential depth of the given example is 3.

In the next section we present a simulation based optimization technique for determining the sequential depth of an FSM.

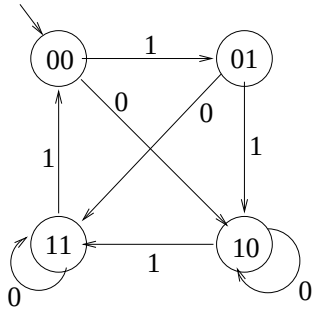

Figure 2. State transition graph

3. Evolutionary Algorithm

In this section the different components of the EA are described. Instead of a single solution, EAs consider a whole set - also called a population. First, the encoding of these elements and their representation is presented in Section 3.1. The "critical part" of the EA is to measure the quality of simulation sequences. This is done in several steps using multi-objective optimization (see Section 3.2). The evolutionary operators used are described in Section 3.3 and finally the overall algorithmic flow - including the detailed choices for the parameter settings - is discussed in Section 3.4.

3.1. Representation

Each individual in the population represents a set of m input vectors $\tilde{Y}$. An upper limit on the length of the vector set is given by the user and the length of one vector is given by the number of input variables k. An individual is a vector set represented by a binary string of length $k \cdot |\tilde{Y}|$.

During the initialization phase, these strings are randomly chosen.

3.2. Objective Function

3.2.1. Simulation

Each individual is evaluated by the objective function to determine its quality. For the evaluation of the objective function the set of vectors represented by an individual is simulated starting from the initial state s_0.

- Starting from the initial state s_0 the set of next states is calculated:

$$S = \bigcup_{i=1}^{|\tilde{Y}|} \delta(\tilde{y}_i, s_0),$$

  where $\tilde{y}_i \in \tilde{Y}$.

- Then for each *new* state in S and $\tilde{Y}$ the set of next states is calculated. I.e.:

$$S = \bigcup_{i=1}^{|\tilde{Y}|} \bigcup_{j=1}^{|S_{new}|} \delta(\tilde{y}_i, s_j),$$

```
compute_depth (individual){
    S := {s_0} ;
    S_new := {s_0} ;
    S_present := {s_0} ;
    depth := 0 ;
    do {
        for i := 1 to |Ỹ| do {
            for j := 1 to |S_new| do {
                s := δ(ỹ_i, s_j) ;
                S := S ∪ {s} ;
            } ;
        } ;
        depth := depth +1 ;
        S_new := S \ S_present ;
        S_present := S ;
    } while (S_new ≠ ∅) ;
    return depth ;
}
```

Figure 3. Objective function

where $y_i \in \tilde{Y}$ and $s_j \in S_{new}$.

- This is repeated, until no *new* state is found.

A sketch of the algorithm is given in Figure 3. The sets S, S_{new} and $S_{present}$ are initialized with the initial state. In set S all states reached during the exploration are included. S_{new} describes only the set of *new* states reached in the present exploration step and $S_{present}$ is set S one time step before. Then for each vector in $\tilde{Y}$ and each state in S_{new} the next states are calculated. If no new state is found the algorithm terminates and the present value of *depth* is calculated by the input set $\tilde{Y}$.

3.2.2. Multi-objective Optimization

For EAs it has been observed that often a single objective function is not sufficient to allow for high quality results. Using only the computed depth as optimization criterion would prefer input vectors that calculate a *maximum* (instead of the sequential) depth of the given FSM. Thus, specialized techniques have been developed following the paradigm of *Multi-Objective Optimization* (MOO) [8].

In our application we make use of the MOO technique proposed in [9] that has been integrated in the software library GAME [11]. For each criterion a priority has to be determined, that ranks "how important" this objective is. The choices for our application are given in Figure 4. As can be seen, two optimization objectives have the highest priority: the total number of reached states has to be maximized and the computed depth has to be minimized. Thus, the input sets are optimized such that a maximum number of states with a minimum depth is reached. Furthermore, objectives with descending priorities maximize the number of states reached in level k, $2 \leq k \leq depth$. Then the input sets where the states are visited "as fast as possible" are preferred.

3.3. Operators

Now the evolutionary operators that are the "core operators" of EA applications are described. First, we distinguish between "standard" crossover operators (well-known for EAs [7]) and problem specific operators [6, 5, 12]. In our framework we only make use of the standard operators and one problem specific "meta operator", that is a generalization of all the others. Additionally, we make use of "classical" mutation operators to explore the local region of proposed solutions.

First, the "standard" EA operators are briefly reviewed: All operators are directly applied to binary strings of length l that represent elements in the population. The parent(s) for each operation is (are) determined by *Tournament*-selection. For the selection of each parent element two individuals are randomly chosen from the population. Then the better individual - with respect to its ranking in the population - is selected.

Crossover: Construct two new elements c_1 and c_2 from two parents p_1 and p_2, where p_1 and p_2 are split in two parts at a cut position i. The first (second) part of c_1 (c_2) is taken from p_1 and the second (first) part is taken from p_2. (Notice, that a special case of this operator is the *horizontal crossover* from [5], where the cut position is chosen only between two test vectors, i.e. test vectors are not split up.)

2-time Crossover: Construct two new elements c_1 and c_2 from two parents p_1 and p_2, where p_1 and p_2 are split in three parts at cut positions i and j. The first (second) part of c_1 (c_2) is taken from p_1 (p_2), the second part is taken from p_2 (p_1) and the last part is again taken from p_1 (p_2).

Uniform Crossover: Construct two new elements c_1 and c_2 from two parents p_1 and p_2, where at each position the value is taken with a certain probability from p_1 and p_2, respectively.

Next, the problem specific operator is presented. The string representation of a sequence of vectors is interpreted as a two-dimensional matrix, where the x-dimension represents the number of inputs and the y-dimension represents the number of vectors. The operator works as follows [12]:

Free Vertical Crossover: Construct two new elements c_1 and c_2 from two parents p_1 and p_2. Determine for each test vector t a cut position i_t. Divide each test vector t of p_1 and p_2 in two parts at cut position i_t. The first (second) part of each test vector of c_1 (c_2) is taken from p_1 and the second (first) part is taken from p_2. (Notice, that the *vertical crossover* from [6] is a special case of this operator, if i_t is equal for all test vectors t.)

priority	objective
1	maximize total number of visited states
1	minimize depth
$k, k = 2, \ldots, depth$	maximize number of visited states in depth k

Figure 4. Optimization objectives and their priorities

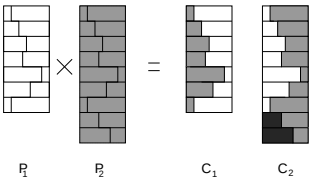

Figure 5. Example for Free-Vertical Crossover

Example 2 The behavior of the *free vertical crossover* is illustrated in Figure 5. The black filled areas result, if vector sets of different size are considered; then, the offsprings are filled with randomly generated values. (But, up to now, in this application all individuals have the same length.)

Moreover, three (standard) mutation operators are applied which are based on bit-flipping at a random position.

Mutation (MUT): Construct one new element c from a parent p by copying the whole element and changing a value at a randomly chosen position i.

2-time Mutation: Perform MUT two times on the same element.

Mutation with neighbor: Perform MUT at two adjacent positions on the same element.

Obviously, all evolutionary operators generate only valid solutions, if they are applied to binary strings.

3.4. Algorithm

We now introduce the basic EA which describes the overall flow. (A sketch is given in Figure 6.)

- The initial population of size $|\mathcal{P}|$ is generated, i.e. the binary strings of length l are initialized using random values.
- Two parent elements are determined by *Tournament*-selection.
- Two new individuals are created using the evolutionary operators with given probabilities.
- These new individuals are then mutated by one of the mutation operators with a fixed mutation rate.

```
approximate_sequential_depth (circuit) {
    generate_random_population () ;
    evaluate_population () ;
    do {
        apply_evolutionary_operators () ;
        evaluate_offsprings () ;
        update_population () ;
    } while (not terminal case) ;
    return (best_element) ;
}
```

Figure 6. Sketch of basic algorithm

- The quality of the elements is determined by simulation and MOO ranking.
- The elements which lost the tournament selection in the present parent population are deleted and the offsprings are inserted in the population.
- The algorithm stops if the best element has not changed for 100 generations.

For the experiments the following parameters have been used: The population size is set to $|\mathcal{P}| = 24$. The vertical crossover is carried out with a probability of 80% and one out of the standard crossover operators is carried out with a probability of 20%, respectively. The offsprings are mutated with a probability of 15% by one of the mutation operators.

4. Experimental Results

The techniques described in the previous section have been implemented using the software library GAME [11]. All algorithms are written in $C/C++$ and the experiments were all run on a SUN Ultra with 256 MByte main memory. As a simulator for evaluation of the objective function we used a simple functional approach based on the ideas of [1]. Here, the underlying BDD package is CUDD from [15]. For the experiments a sample of the benchmarks from ISCAS were taken.

The experimental results are given in Table 1. The name of the benchmark is given in the first column. The columns *Sim* and *SAT* give the results from [16] and [13], respectively. It is important to notice that *Sim* obtains estimations only, while *SAT* give the exact numbers. As can be seen, compared to *SAT* the other technique gives over-

Table 1. Experiments for ISCAS circuits

name	Sim	SAT	EA
s298	18	18	18
s208	255	n.a.	255
s349	6	n.a.	6
s386	n.a.	7	7
s499	n.a.	21	21
s510	46	n.a.	46
s526	150	n.a.	150
s641	n.a.	6	6
s713	10	6	6
s820	n.a.	10	10
s953	n.a.	10	10
s1196	5	2	2
s1488	21	21	21

as well as under-approximations, what makes them hard to use in real-world scenarios.

The results of our EA approach are given in the last column. It can be observed that in all cases the exact results (where this is known) is computed. But since the EA is based on simulation, it can also be applied to larger circuits.

In this way, the presented technique combines the best of [16] and [13], i.e. very high-quality results are computed but for the evaluation no time consuming proof techniques, like BDD or SAT, are used. Instead, simulation that can be carried out in linear time in the circuit size is successfully applied.

5. Conclusions

In this paper a simulation-based approach for the computation of the sequential depth of a FSM has been presented. Due to the choice of the objective function results of high quality can be obtained. This finds direct application in BMC, since the depth of the FSM corresponding to the DUV gives the results for the maximal time interval that has to be considered.

The run time of the algorithm is dominated by the simulation time. For this, it is a focus of current work to integrate a more efficient parallel simulator in the GAME software library.

References

[1] P. Ashar and S. Malik. Fast functional simulation using branching programs. In *Int'l Conf. on CAD*, pages 408–412, 1995.

[2] B. Bentley. Validating the Intel Pentium 4 microprocessor. In *Design Automation Conf.*, pages 244–248, 2001.

[3] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conf.*, pages 317–320, 1999.

[4] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Sequential circuit verification using symbolic model checking. In *Design Automation Conf.*, pages 46–51, 1990.

[5] F. Corno, P. Prinetto, M. Rebaudengo, and M.S. Reorda. GATTO: A genetic algorithm for automatic test pattern generation for large synchronous sequential circuits. *IEEE Trans. on CAD*, 15(8):991–1000, 1996.

[6] F. Corno, P. Prinetto, M. Rebaudengo, M.S. Reorda, and R. Mosca. Advanced techniques for GA-based sequential ATPG. In *European Design & Test Conf.*, pages 375–379, 1996.

[7] L. Davis. *Handbook of Genetic Algorithms*. van Nostrand Reinhold, New York, 1991.

[8] K. Deb. *Multi-objective Optimization using Evolutionary Algorithms*. John Wiley and Sons, New York, 2001.

[9] N. Drechsler, R. Drechsler, and B. Becker. A new model for multi-objective optimization in evolutionary algorithms. In *Int'l Conference on Computational Intelligence (Fuzzy Days)*, volume 1625 of *LNCS*, pages 108–117. Springer Verlag, 1999.

[10] R. Drechsler and N. Drechsler. *Evolutionary Algorithms for Embedded System Design*. Kluwer Academic Publisher, 2002.

[11] N. Göckel, R. Drechsler, and B. Becker. GAME: A software environment for using genetic algorithms in circuit design. In *Applications of Computer Systems*, pages 240–247, 1997.

[12] M. Keim, N. Drechsler, R. Drechsler, and B. Becker. Combining GAs and symbolic methods for high quality test of sequential circuits. *Jour. of Electronic Testing: Theory and Applications*, 17:141–142, 2001.

[13] M. Mneimneh and K. Sakallah. SAT-based sequential depth computation. In *ASP Design Automation Conf.*, 2003.

[14] E.M. Rudnick, J.H. Patel, G.S. Greenstein, and T.M. Niermann. Genetic algorithm framework for test generation. *IEEE Trans. on CAD*, 16(9):1034–1044, 1997.

[15] F. Somenzi. Efficient manipulation of decision diagrams. *Software Tools for Technology Transfer*, 3(2):171–181, 2001.

[16] C.-C. Yen, K.-C. Chen, and J.-Y. Jou. A practical approach to cycle bound estimation for bounded model checking. In *Int'l Workshop on Logic Synth.*, pages 149–154, 2002.