How Much Randomness Makes a Tool Randomized?

Petr Fiser, Jan Schmidt
Faculty of Information Technology, Czech Technical University in Prague
Prague, Czech Republic
fiserp@fit.cvut.cz, schmidt@fit.cvut.cz

Abstract—**Most of presently used academic logic synthesis tools, including SIS and ABC, are fully deterministic. Up to the knowledge of the authors, this holds for all available commercial tools as well. This means that no random decisions are made; the algorithms fully rely on deterministic heuristics. In this paper we present several hints of insufficiency of such an approach and show examples of perspective randomized logic synthesis algorithms. Judging from our experiments, these algorithms have a higher potential of performing better than the deterministic ones. Further we study how much randomness is actually needed for the algorithms to perform well. We show that some algorithms require only a small amount of randomness, while still taking full advantage of their randomized nature. On the other hand, some algorithms require a very high level of randomness to perform well. We propose reasons for this behavior and show a way of computing the necessary measure of randomness required.**

Keywords-logic synthesis, minimization, randomized algorithms, derandomization, Espresso, ABC

I. INTRODUCTION

Basic principles of most of viable logic synthesis algorithms have been established already in early 1960's, both for two-level (PLA) minimization [1], [2] and multi-level logic synthesis [3], [4], [5], [6], [7]. These basic algorithms suffer from low scalability for large designs. This drawback has been partially overcome by introducing binary decision diagrams (BDDs) [8], [9]. New, BDD-based algorithms were proposed [10], [11], [12]. However, even though BDDs represent logic functions implicitly, their size may easily blow up exponentially as well [8], [9].

In two-level minimization, the basic Quine-McCluskey algorithm [1], [2] has been replaced by Espresso [13], which became a well-established standard since 1980's.

These logic synthesis algorithms were implemented in academic tools MIS [14], SIS [15], and MVSIS [16] by Berkeley Logic Synthesis and Verification Group.

Recently, the research shifted towards a different internal network representation: the And-Inverter Graphs (AIGs) [17]. These are more scalable than standard tabular (PLA) circuit nodes representations and new, more flexible synthesis and mapping algorithms may be applied upon these structures [17], [18], [19], [20]. A synthesis tool ABC [21] implementing these algorithms came as a successor of SIS and MVSIS and its development still continues.

Because of high complexities of present designs, using exact algorithms (for two-level minimization, multi-level optimization, resynthesis, etc.) is not feasible. Therefore, approximate heuristic algorithms must be used in practice, to reduce the search space. Even though the employed heuristics usually produce solutions of a sufficient quality, they do not guarantee an optimum and mostly do not even guarantee a maximum relative error.

All of the mentioned algorithms and logic synthesis systems (SIS, ABC) are fully deterministic; no random choices are made in the synthesis process. This brings a benefit of reproducibility of the result – two runs of the algorithm using the same data will produce equal results. However, the determinism also involves inability of reaching different, possibly much better results.

The logic optimization can be treated as a general combinatorial problem. In general, most of algorithms used in logic synthesis are of a *local search* nature. Here moves in the state space are usually driven by a deterministic heuristic function. For example, when checking for tautology (which is a co-NP-hard problem) in Espresso [13], the variable splitting decision is made on the most binate variable. It has been shown experimentally, that such an approach is "good". However, such a conclusion definitely cannot be generalized for any tautology-checking problem instance.

Moreover, a *granularity* of the heuristic function may become of concern as well. This means, it may happen that there will appear two or more *equally* valued possibilities for moves in the state space. Mostly this happens in cases where the knowledge of the state space is limited, the heuristic function cannot be evaluated precisely enough, or simply there are several equivalent moves. In the Espresso tautology checking example, it may happen that two or more variables that are most and equally binate will appear. Then there is no clue which one to select, so the first one found is chosen. However, the decision on one variable may significantly affect the algorithm run.

Basically, there are two ways of solving this problem: either a parallelism is involved, or randomized algorithms are used. In the case of parallelism used, several heuristic decisions are made simultaneously. From the theoretical point a view, the problem is solved by a non-deterministic Turing machine, which can be simulated on a sequential machine by, e.g., the best-first search. Here the number of simultaneously active states may grow exponentially, thus the general model cannot be implemented for practical problem instances. Here, e.g., modifications of the best-first search strategy, like the

beam-search [22], [23] are used. Approaches based on genetic algorithms partially belong to the category of parallel search algorithms as well [27], [28].

The disadvantage of such an approach is that some perspective parts of the state space may not be explored, due to state explosion (i.e., only a strictly limited number of states can be stored as candidates for expansion).

In the case of randomization employed, one of the "equivalent" moves is chosen randomly, hoping that a "good path" through the state space will be found. In a repeated run of the randomized process, different choices will be (hopefully) selected. As a result, larger part of the search space is explored, at the cost of runtime.

In the limit case (unlimited space and time resources), these two strategies produce equal results. Theoretically, repeatedly restarted randomized algorithm fully simulates the non-deterministic Turing machine.

There have been several attempts to use randomized algorithms in logic synthesis. The optimization algorithm proposed in [24] is deterministic, though it is suggested to be run repeatedly, with random initial solutions. The local optimum is avoided this way.

Randomized algorithms based on simulated annealing [25], [26] or genetic algorithms [27], [28] were proposed for logic synthesis. Since these algorithms are of an incremental nature, a tradeoff between the result quality and runtime may be set. Any combinatorial problem can be solved in a randomized way using these problem-independent metaheuristics, as a universal way to trade time for quality. The question is, can *problem-dependent* randomized algorithms be designed, with better efficiency.

Recently there has been developed a logic optimization method based on a Cartesian Genetic Programming (CGP) [29], [30]. This algorithm is able to reach outstandingly good results (compared to, e.g., SIS or ABC), at the cost of long runtime. Alike in all evolutionary algorithms, randomness is the basis of success here.

In the past we have also proposed several synthesis algorithms where randomness is essential for their successful execution [32], [33], [34]. Some of these algorithms will be briefly reviewed in Section III, for the sake of understanding the rest of the paper.

We present some reasons for using randomized algorithms in logic synthesis and show several synthesis processes, where randomness becomes beneficial. Randomized and de-randomized algorithms are compared to justify our claims.

The main scientific contribution of this paper is the discussion on the required measure of randomness, i.e., the quality of the random number generator used. These measures are evaluated for different randomized processes. We show that some randomized algorithms do not require too high level of randomness; for example, we have observed that a random number generator producing only two different values is sufficient, in order to make a particular algorithm "fully randomized". Conversely, a high level of randomness is required for some algorithms, in order to perform well. Possible ways of explanation of this phenomenon will be presented and a way of computing the upper bound of the necessary measure of randomness will be proposed.

II. MOTIVATION

A. Design Variety

As it was said in the introduction, deterministic algorithms benefit from reproducibility of the result, but they may suffer from "insufficiency" of the heuristics used or inability to explore sufficiently large search space due to resources limitations. As present circuits become larger, also the variety of features in a single design increases. We have shown logic synthesis examples (benchmarks), which are "difficult" for conventional synthesis processes [35], [36]. The "difficulty" of these circuits consists in specificity of design processes that have to be conducted in order to reach satisfactory results. Heuristics that have been found successful for most of designs of 1980's need not be that successful for present designs. Altering these heuristics to *adapt* to any design is probably impossible. Here randomized algorithms could help – the performance need not be optimum, however there will be still a non-zero chance of reaching the desired solution. The CGP-based optimization process is such a case [29], [30].

B. Iterative Synthesis

The concept of iterating the logic synthesis process was proposed in Espresso [13]. Here the two-level minimization process is conducted in two nested loops, where each loop is terminated when no improvement is obtained, with respect to the previous iteration. Even though techniques to get out of the local minimum are employed, the overall strategy belongs to the "best-only" search. All Espresso algorithms are deterministic, thus for some "well-tailored" benchmarks Espresso could fail.

A kind of an iterative process was proposed in ABC [21], for a multi-level synthesis followed by technology mapping. Here the authors suggest running the sequence of resynthesis and technology mapping commands repeatedly, to reach better results. The network is iteratively refined by this way, partially respecting demands of the mapper. However, since all ABC algorithms are deterministic, such an iterative process will quickly get stuck in a local optimum [34].

Randomness in the iterative synthesis could introduce new structures, that may (or may not) be beneficial. Moreover, a higher level of randomness (say, kind of *mutations*) could introduce structures that couldn't have been produced by a deterministic algorithm. Introducing "accidentally bad" structures should not matter in the iterative synthesis; they can be just thrown off. Or, parts of them could contribute to the solution. Chatterjee and Mishchenko proposed the concept of "*structural choices*" [38], which is now implemented in ABC. Here the structure of the network is not forgotten after one synthesis step, all network structures that ever appeared during the synthesis process are accumulated, and later exploited in technology mapping.

C. Sources of Randomness in Deterministic Algorithms

EDA tools designers sometimes object to randomization by claiming that their algorithms *cannot* be randomized, for their very nature. The algorithms always choose the (locally) best decision under given circumstances and therefore there is no place for random decisions. However, we will show that this needs not be true.

The first source has already been mentioned in the introduction. If the "best-only" local heuristic is used, and there are multiple "best" moves, the result depends on lexicographic order of the moves examined. In the case of the "first improvement" heuristic, the situation can be much worse if the moves are examined in some fixed lexicographical order; the search is biased towards moves located earlier in the order. Thus, the sensitivity of results to the (semantically equivalent) ordering of input data is an indication of lexicographical bias.

To present an example, Espresso [13] is *sensitive to permutation of variables.* This means that when the columns in the PLA file (which is an input to Espresso) are permuted, Espresso returns different results. Of course, the permuted PLAs are functionally equivalent, just the ordering of variables in the matrix is different (the equivalence was verified by the –DPLAverify Espresso command). We have processed 138 PLAs from the MCNC benchmark set [39], each PLA 10,000 times, while variables of each were randomly permuted. As a result, we have observed up to 43% difference in the literal count between the minimum and maximum obtained minimized PLA. The average difference was 2.1%. Results of five of the most striking PLAs are shown in TABLE I. After the circuit name, the number of literals of the unmodified minimized benchmark circuit is shown ("*Orig.*"). Then minimum, maximum and average literal counts from the 10,000 permutations (after minimization) are shown. Finally the percentage difference between the minimum and maximum values is given. The data of all the 138 circuits are summed (averaged) in the last table row.

The results indicate that some essential algorithms in Espresso process variables in a lexicographical order. Thus, the minimization results may be improved by just properly permuting the inputs. For example, results of the original *dk48* and *pdc* benchmarks are very close to the worst case permutation; they can be improved at least by 20%. In the extreme cases, the circuit size was increased (*Orig. vs. Max*).

Surprisingly, the result quality differs in Espresso-Exact as well, by up to 6% of literals (the numbers of SOP terms are, of course, equal).

TABLE I. INFLUENCE OF VARIABLE ORDERING IN ESPRESSO

Circuit	*Orig.*	*Min*	*Max*	*Avg*	*Diff.*
mark1	97	85	149	102.9	43.0%
dk48	115	92	124	106.3	25.8%
pdc	912	727	961	794.0	24.3%
ex5	444	399	463	435.8	13.8%
z5xp1	287	260	295	267.0	11.9%
Sum/Avg.	157,412	156,434	158,599	157,283	2.1%

The average quality difference in the Espresso case is not that remarkable. However, we have studied many processes in ABC [21] and we have found that most of them are not immune to variable ordering in the source file as well. The source file (BLIF [40]) describes a logic network. Here the names of the network primary input and output variables are listed in the file header. When we changed the order of these variables in the header (which does not alter the circuit function, or the meaning of variables), ABC produced different results, sometimes significantly differing in quality.

We have processed 228 circuits from the IWLS'93 benchmark set [41]. The experimental setup was the same as in the Espresso case: each circuit was processed 10,000-times with different variable ordering. We have observed the sensitivity on variable ordering even for the most basic ABC command for balancing the AIG, "*balance*". Even here the maximum difference in the resulting circuits' sizes was more than 10% (measured in the number of AIG nodes).

Finally we have simulated a standard synthesis process: the circuits were optimized by the "*choice*" script and mapped into standard cells ("*map*"). Samples of results are shown in TABLE II. The format is retained from TABLE I.

We can see that for the circuit "*squar5*" there is more than 67% difference between "worst" and "best" variable ordering. The average difference was more than 11%.

TABLE II. INFLUENCE OF VARIABLE ORDERING

Circuit	*Orig.*	*Min*	*Max*	*Avg*	*Diff.*
b12	372	184	566	387.6	67.5%
squar5	50	35	90	53.8	61.1%
z4ml	75	60	140	92.3	57.1%
rd53	65	37	73	55.4	49.3%
x4	333	285	445	348.8	36.0%
apex5	1015	946	1248	1093.0	24.2%
Sum/Avg.	135,476	130,664	140,865	135,544	11.0%

A similar sensitivity to variable ordering may also be observed in the BDD-based decomposition tool BDS [11]. Here the default BDD variable ordering is given by lexicographic order of input variables in the source BLIF.

Variable ordering is one of the points where randomness can be introduced without affecting principles of the algorithms used. There is a chance that different results will be obtained when variables will be randomly reordered prior to the execution of the algorithm. The overall result quality may be improved by repeatedly running the synthesis and picking the best result obtained, or in an iterative way described in Subsection II.B.

D. Result Reproducibility

The main objection of EDA tools designers to randomization is the problem of reproducibility of results. A randomized algorithm will return different results when run repeatedly on the same data. This is not desirable because of several aspects:

1. Unexpected behavior can occur when re-synthesizing complex designs.
2. Design errors will be difficult to locate, assuming the synthesis will always produce different solutions.
3. Small changes in the specification shall produce small changes in solution.

Problems ad 1 and 2 can be easily avoided by fixing the random number generator *seed* via a synthesis parameter (together with many other parameters, like, e.g., optimization effort). This will not affect the randomized nature of the algorithms, and equal results will be obtained whenever required.

For problems ad 3, we must assume that the algorithm will be used in a way that will produce nearly-optimum results, stabilizing the solution. Of course, cases can be constructed where adding a single term turns a large function into tautology, etc.

Therefore, we do not consider the result reproducibility as a serious threat.

III. Randomized Logic Synthesis Algorithms

In this section we will briefly review some randomized algorithms developed in the past and show the importance of randomness. In particular, we will compare results obtained by the original randomized algorithm and de-randomized ones. De-randomization was done by modifying the random number generation function, so that it produces only 1, 2, etc. distinct values. In further text, the measure of randomness will be denoted as *RF* (*randomness factor*). For *RF* = 1, the degraded random number generator always produces one value, constant 0. For *RF* = 2, two border values are produced (0, MAXINT), etc. For *RF* = *infinity* the unmodified random number generator is used.

We will show examples where a high level of randomness is essential for a successful algorithm run and examples where high randomness, though beneficial, is not essentially important.

All the presented randomized algorithms are of an iterative nature – the final result is obtained by iteratively refining an intermediate solution. Intermediate solutions are then obtained based on random decisions.

A. Two-Level Minimizer BOOM

The idea of partially randomized generation of the solution is used in a two-level (SOP) minimizer BOOM [31], [32]. The minimization is run repeatedly, whereas a new set of implicants covering the source function is produced in each iteration.

In general, BOOM comprises of four consecutive steps: *CD-search*, *implicant expansion*, *implicant reduction*, and *covering problem solution*. CD-search (Coverage-Directed search) is the vital phase of BOOM. Here implicants are generated by reducing a universal hypercube (*n*-dimensional cube, where *n* is the number of variables) by adding literals one by one to it. This is conducted until the term becomes an implicant, i.e., stops intersecting the off-set. Then the implicant is stored and a next one is generated, until the whole on-set is covered.

We use a simple, but efficient heuristic to select literals for addition to the term under construction: the cost function is the frequency of appearance of a given literal in the uncovered on-set. Literals maximizing this cost function are selected into the solution. However, it often happens that the cost is equal for several literals. In this case, one of these literals is selected *randomly*. As a result, repeated runs of CD-search may produce different results, i.e., different sets of terms covering the on-set.

To introduce even more randomness to the implicant generation process, *mutations* may be present. With a given probability, a mutation occurs. Then a literal with any non-zero cost is selected, instead of the literal with the maximum cost. We have found experimentally, that 2-5% of mutations are beneficial. For details see [42].

The obtained terms are further expanded to prime implicants and then reduced to obtain group implicants. These two phases are randomized as well; the direction of expansion/reduction is chosen randomly.

The implicant generation phase is iterated for a given number of cycles and all produced implicants are stored in one common implicant pool and the covering problem using all implicants is solved, to find an irredundant cover. Only new implicants are recorded in the pool, so that no duplicities occur. The basic BOOM algorithm is shown in Figure 1.

```
BOOM(F, R) {        // F = on-set, R = off-set
    Pool = ∅;
    do {
        Cover = CD-Search(F, R);
        Pool = Pool ∪ Cover;
        Pool = Pool ∪ Expand(Cover, R);
        Pool = Pool ∪ Reduce(Cover, R);
    } while (!stop());
    Solution = CP_Solve(F, Pool);
    return Solution;
}
```

Figure 1. BOOM algorithm

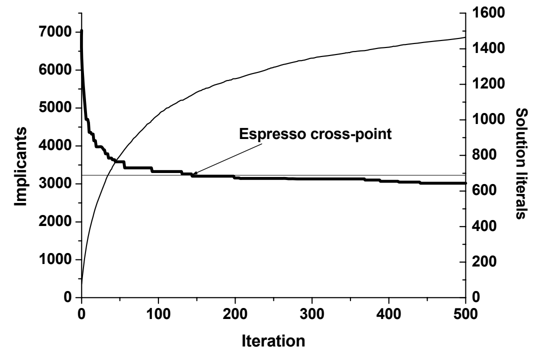

Figure 2. Iterative process in BOOM

The benefit of iterating the process is illustrated in Figure 2. A randomly generated function of 20 input variables, 5 output variables, 200 care terms and 10% of input don't cares was processed. A random function was chosen in order to maximally suppress an influence of any possible singular behaviors of industrial benchmark circuits.

The graph shows the progress of the minimization, in terms of the total number of implicants in the pool and the solution quality (CP was solved after each iteration, for the example purposes). The solution quality, in terms of total sum-of-products literals is depicted by the bold line (and the right y-axis). We can observe that the number of implicants follows the saturation curve, while the solution improves in the progress. The deterministic result obtained by Espresso [13] is shown as a horizontal hairline. It can be seen that even though rather inferior solutions are produced in the early iterations, BOOM overcomes Espresso in the solution quality in the 144-th iteration. This result may be generalized for any circuit. In cases where Espresso does not produce exact results, BOOM is able to obtain them for a possible cost of runtime.

The importance of randomness in the minimization process is illustrated in Figure 3. Here BOOM was de-randomized, as described in the beginning of this Section and the progress of the implicants number growth was traced. The final result quality obtained after 1000 iterations for different *RF*s is shown in TABLE III. and the progress of the result quality during 1000 iterations is visualized by Figure 4. The values were obtained by averaging 5 BOOM runs (for each *RF* value).

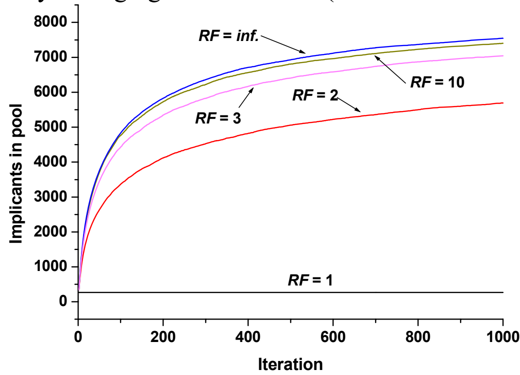

Figure 3. Derandomized BOOM – implicant growth

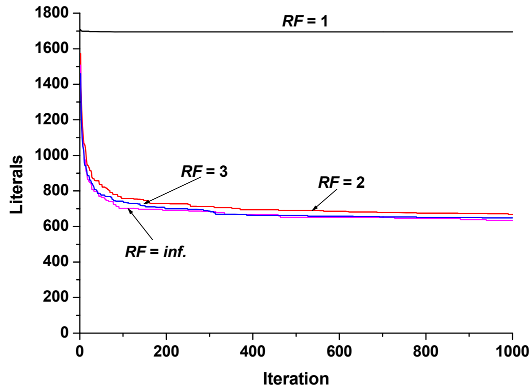

Figure 4. Derandomized BOOM – result quality

TABLE III. DERANDOMIZED BOOM – RESULT QUALITY

RF	*Literals*
1	1695
2	669
3	650
10	648
100	649
infinity	647

We can see that when the capabilities of the random number generator are limited, the number of generated implicants grows slower and the solution quality drops as well. For *RF* = 1 the iterative process is not working at all, since equal implicants are generated in each iteration.

But even for *RF* = 2 the implicant generation rate starts to follow the saturation curve and for *RF* = 3 the rate nears the rate of *RF* = *infinity*. For *RF* > 10 there is no noticeable difference from the fully randomized algorithm. Regarding the result quality, *RF* = 1 definitely lacks here. For *RF* > 1 there are only slight differences in quality.

The above observations can be backed up by the fact that in CD-search there are usually only few "equal" choices to decide between. A histogram and a pie-chart of the distributions of the number of choices (for our example circuit, fully randomized algorithm run, and 200 iterations) are shown in Figure 5. In 40% of cases there is only one option to choose from. There are 2 choices in less than 20% of cases, and the distribution curve sinks exponentially. The average number of choices was 3.35.

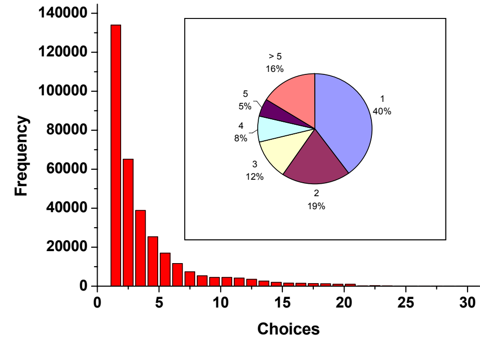

Figure 5. Numbers of choices

From a theoretical point of view, the maximum number of possible choices in every step equals to the number of different function's literals, i.e., twice the number of input variables, which is 40 in our case. However, the maximum of choices encountered in our example was 31 only.

Concluded, BOOM needs not too much of randomness for its successful run. Even for functions with a higher number of variables, the number of possible decisions cannot reach millions. Let us note here that *perfect* random number generator is considered; even though we claim that generating only $2n$ different random numbers (where n is the number of input variables) is sufficient, the real random number generator must obviously have much more than $2n$ internal states.

B. *Randomized Multi-Level Resynthesis*

In [34] we have proposed a multi-level resynthesis method, where the network is iteratively processed by parts, in contrast to resynthesizing the network as whole (as proposed by authors of ABC, see Subsection II.B). The pseudo-code of the basic algorithm is shown in Figure 6. The network is iteratively refined by extracting a "*window*" in the network and resynthesizing it by ABC. Generally, the window is a *connected* part of the network of a user-specified size. The limit case, where the window size equals to the whole network size, equals to the suggested iterative process in ABC.

```
Resynthesize(Network N) {
    do {
        W = Extract_Window(N);
        W' = resynthesize_by_ABC(W);
        N' = (N-W) ∪ W';
        if (cost(N') ≤ cost(N)) N = N';
    } while (!stop());
}
```

Figure 6. Resynthesis by parts algorithm

There were several window extraction strategies proposed [34]. In this paper we will present only one representative: the *Radius extraction*. First, a *pivot* node (gate) is selected randomly in the network. Then nodes reachable in a given distance (radius) from the pivot are moved to the window. In particular, transitive fan-in and fan-out nodes of the pivot are selected, up to a given radius.

Even though the idea of resynthesis by parts could seem to be obviously less efficient than resynthesis of the whole circuit (e.g., global information of the circuit structure is missing during the resynthesis), it is not. Most probably the only reason for the efficiency is the introduction of randomness; randomness will help the iterative optimization process escape from a local optimum. An illustrative example is shown in Figure 7. for the IWLS'93 benchmark "*e64"* [41]. The circuit was iteratively resynthesized by ABC as whole (see the "100% resynthesis" curve) and by the iterative resynthesis by parts, using Radius extraction, radius 5 (see the *RF* = *inf.*" curve). We have used the ABC "*choice*" script followed by "*map*" [21] as the resynthesis procedure. The curves were obtained by averaging 20 resynthesis runs.

It can be seen that the repeated resynthesis of 100% of the circuit quickly converged to a local minimum. Conversely, the randomized resynthesis method converges slower, but quickly reaches much better results. Such a behavior was observed for a vast majority of examined circuits [34].

Like in the previous Subsection, we have investigated the influence of the random factor on the process. Convergence curves for the "*e64"* [41] circuit are shown in Figure 7. as well. Here we see that for *RF* = 1 the process is rather insufficient and quickly converges to a local minimum, which is even worse than that of 100% resynthesis. However, even for *RF* = 2 the convergence curve nears the *RF* = *inf.* one, for *RF* = 100 the curves blend (not shown in the Figure).

All the data was obtained by averaging results of 20 independent runs, to make the results precise.

The necessary measure of randomness can be derived analytically as well. The random choice occurs in the pivot selection procedure. Here the number of choices equals to the number of the network gates. Thus, the number of the initial network gates is the upper bound of the number of different values the random number generator needs to produce.

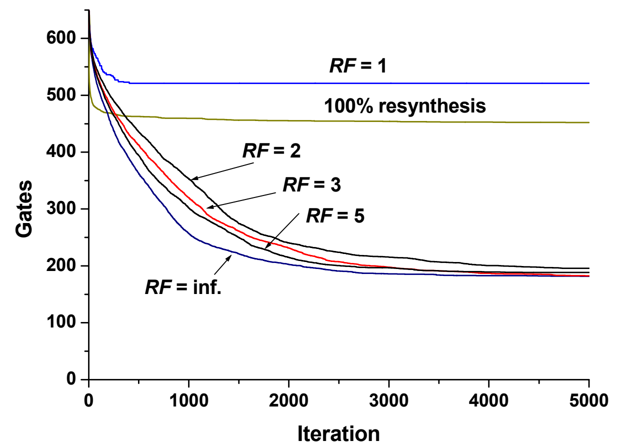

Figure 7. Resynthesis by parts – derandomized

C. *FC-Min: Probabilistic Two-Level Minimization*

In [33] we have presented a two-level minimizer primarily targeted to multi-output functions. Basically, group implicants are generated directly from the PLA description [13] of the function, by looking for a *rectangle cover* [43], [7] of the matrix defining output values of the multi-output function. In particular, rectangles comprising of maximum of '1' are looked for. A randomized greedy heuristic is employed to solve this NP-hard problem. First, a matrix row having most of 1's is selected as a starting point. Then rows maximizing the size of the rectangle are gradually appended, until its size cannot be further increased. Implicants are then derived as supercubes of terms included in the rectangle. The more rows the rectangle has, the higher is the dimension of its respective term, since supercubes of more terms are produced. As a consequence, rectangles spanning many rows more likely induce terms that intersect the function's off-set, therefore they cannot be parts of the solution, so they are discarded and a different rectangle is looked for. One way to overcome this problem is a *probabilistic execution* of the rectangle generation process: in each step, the rectangle generation is stopped with a probability given by a parameter called the *depth factor* (*DF*). The higher *DF* is, the more likely will the algorithm continue increasing the number of rows. The pseudo-code of the rectangle generation algorithm is shown in Figure 8. The input to the algorithm is the output matrix of the PLA, the output is one rectangle covering some of the matrix '1's. The randomized termination condition is visualized in bold.

```
FindRectangle(O) { //O is the output matrix (m, p)
   R = ∅;              // empty row set
   C = ∪{0, ..., m};   // set of all columns
   do {
      v = row_with_maximum_x_for(0 ≤ i < p)
          where x = (|R|+1)*|C ∩ O[i]| - |R|*|C|;
             // potential increase of covered '1's
      if ( v < 0 ) break;
         // no further increase possible. Terminate
      R = R ∪ {v};  // include v into C
      C = C ∩ O[v]; // reduce C
   } while (random() < DF);
         // forced random termination
   return (R, C);
}
```

Figure 8. Find rectangle algorithm

Random nature of this algorithm guarantees that the search will ever stop. Decreased randomness decreases the variety of implicants generated by FC-Min. When FC-Min is run iteratively in a BOOM-like way (see Subsection III.A), this will involve a reduced implicant growth rate. However, since the algorithm termination condition is *continuous* (`random()` generates real numbers here), much higher level of randomness is required for a successful algorithm run. This is documented in Figure 9. The function from Subsection III.A was minimized (20 inputs, 5 outputs, 200 terms), *DF* was set to 0.8. The values were obtained by averaging 20 runs. The growth of the number of implicants during 1000 iterations, for different *RF*s is shown. We can see that even for *RF* = 100 the implicants number grows rather slowly, compared to *RF* = *infinity*. For *DF* = 1 the algorithm got stuck, which is expectable (the stopping condition is never satisfied).

The solution quality is affected in the same way. *RF* of at least 1000 is required, in order to approach the solution of the fully randomized process. The progress of the solution quality is depicted in Figure 10.

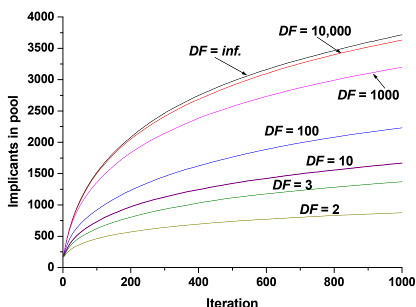

Figure 9. Derandomized FC-Min – implicants number growth

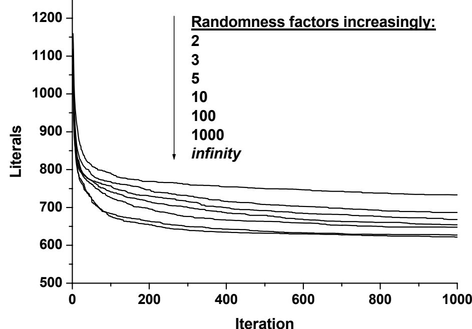

Figure 10. Derandomized FC-Min – solution quality

In contrast to the algorithms mentioned in the previous subsections, in the case of FC-Min the minimum required degree of randomness cannot be analytically computed. In fact, any loss of randomness involves a loss of efficiency here.

D. Simulated Annealing Based Algorithms

The difficulties in analytical estimation of necessary randomness factor can be illustrated by the case of simulated annealing.

The core condition accepting a worsening move can be written as

```
accept = (random() < exp(-delta / (maxdelta*T)))
```

where `delta` is the difference in cost, `maxdelta` an instance-specific upper bound for such difference, and `T` is the current temperature. Notice that this formula is already normalized with respect to the cost function range.

The simulated annealing procedure is known to be robust with respect to variations in the above formula [44], [45], and even to approximation of the exponential function [46]. Therefore, we would expect it to work even with a coarse-grained random generator.

Obviously, the accepting formula is not and cannot be normalized with respect to temperature, and therefore a randomness factor adequate for initial high temperature would be too coarse at the end of annealing. We have to admit that although we can still characterize the measure of randomness by a parameter, the actual values may vary during the computation.

IV. CONCLUSIONS

The contribution of this paper is threefold: first, we have shown to what extend the result of some deterministic algorithms (Espresso, ABC) depends on lexicographical ordering of variables in the source file. Therefore, either more sophisticated heuristics should be developed, or randomness could be employed, in order to possibly improve the result quality. Let us note that the first alternative is very difficult to be accomplished in general: for example, finding an optimum BDD variable ordering is NP-hard itself.

Next we have presented several examples of randomized logic synthesis processes, in opposition to the presently used fully deterministic ones. Some reasons for randomization were proposed and backed up by experimental results showing inefficiency of deterministic algorithms and benefits of randomization.

Finally, the necessary measure of randomness was discussed. We have shown that mostly a negligible amount of randomness is needed for the algorithm to perform well. However, in some cases, any de-randomization may deteriorate the solution. Probabilistic processes like FC-Min [33] or, e.g., algorithms based on simulated annealing [24] are examples.

We have examined the behavior of the presented heuristics using many different example circuits. All the conclusions drawn from the behavior of the example circuits presented in this paper can be generalized for almost any circuit.

ACKNOWLEDGMENT

This research has been supported by MSMT under research program MSM6840770014 and by the grant GA102/09/1668.

REFERENCES

[1] W. V. Quine, "The problem of simplifying truth functions", Amer. Math. Monthly, 59, No.8, 1952, pp. 521-531

[2] E. J. McCluskey, "Minimization of Boolean functions", The Bell System Technical Journal, 35, No. 5, Nov. 1956, pp. 1417-1444

[3] R. L. Ashenhurst, "The decomposition of switching functions". In Proceedings of the International Symposium on the Theory of Switching, Part I 29, pages 74–116, 1957.

[4] H. A. Curtis, "A New Approach to the Design of Switching Circuits". Van Nostrand, Princeton, N.J., 1962.

[5] J. P. Roth and R. M. Karp, "Minimization over boolean graphs," IBM J. Res. Dev., pp. 227–238, Apr. 1962.

[6] G. D. Hachtel and F. Somenzi: "Logic Synthesis and Verification Algorithms", Kluwer Academic Pub, 1996, 564 p.

[7] S. Hassoun and T. Sasao, "Logic Synthesis and Verification", Boston, MA, Kluwer Academic Publishers, 2002, 454 p.

[8] S. B. Akers, "Binary decision diagrams", IEEE Transactions on Computers, vol. C-27, No. 6, June 1978, pp. 509-516

[9] R. E. Bryant, "Graph based algorithms for Boolean function manipulation", IEEE Trans. on Computers, vol. 35, No. 8, August 1986, pp. 677-691

[10] K. Karplus, "Using if-then-else DAG's for multi-level logic minimization," Univ. California, Santa Cruz, UCSC-CRL-88-29, 1988

[11] C. Yang and M. Ciesielski, "BDS: A BDD-Based Logic Optimization System", IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 21, no. 7, pp. 866-876, 2002

[12] O. Coudert and J.C. Madre, "Implicit and Incremental Computation of Primes and Essential Primes of Boolean functions", Proc. of 29th DAC, Anaheim CA, USA, June 1992, pp. 36-39

[13] R. K. Brayton et al., "Logic minimization algorithms for VLSI synthesis", Boston, MA, Kluwer Academic Publishers, 1984, 192 pp.

[14] R. K. Brayton, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert R.Wang, "MIS: A Multiple-Level Logic Optimization System". IEEE Trans. on Computer-Aided Design, pp. 1062–1081, Nov. 1987.

[15] E.M. Sentovich et al., "SIS: A System for Sequential Circuit Synthesis", Electronics Research Laboratory Memorandum No. UCB/ERL M92/41, University of California, Berkeley, CA 94720, 1992

[16] M. Gao, Jie-Hong Jiang, Y. Jiang, Y. Li, S. Sinha, and R.K. Brayton, "MVSIS", In the Notes of the International Workshop on Logic Synthesis, Tahoe City, June 2001

[17] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis". In 43th Design Automation Conference, San Francisco, CA, USA, 2006, pp. 532-535.

[18] A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to technology mapping for LUT-based FPGAs". IEEE Trans. on Computer-Aided Design, Vol. 26(2), Feb 2007, pp. 240-253.

[19] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts", Proc. ICCAD'07, pp. 354-361.

[20] A. Mishchenko, R. K. Brayton, and S. Chatterjee, "Boolean factoring and decomposition of logic networks", Proc. ICCAD'08, pp. 38-44.

[21] Berkeley Logic Synthesis and Verification Group, "ABC: A System for Sequential Synthesis and Verification". [Online].

[22] W. Zhang, "State-space search: Algorithms, complexity, extensions, and applications", Springer: New York, 1999

[23] L. Jozwiak, "Advanced AI Search Techniques in Modern Digital Circuit Synthesis". *Artif. Intell. Rev.* 20, 3-4 (December 2003), 269-318.

[24] D. Brand, "Hill climbing with reduced search space", IEEE International Conference on Computer-Aided Design, 7-10 Nov., 1988 (ICCAD-88), pp. 294-297.

[25] J.M. Sanchez and J. Lanchares, "Multilevel logic synthesis using algorithms based on natural processes", 20th International Conference on Microelectronics, 12-14 Sep 1995, pp. 823-828.

[26] A. Kuehlmann, P. Färm, and E. Dubrova, "Logic optimization using rule-based randomized search", 05 Asia and South Pacific Design Automation Conference, Shanghai, China, 2005, pp. 998-1001

[27] R. Vemuri, "Genetic synthesis: performance-driven logic synthesis using genetic evolution", First Great Lakes Symposium on VLSI, 1-2 Mar, 1991, pp. 312-317

[28] K. Ohmori and T. Kasai, "Logic synthesis using a genetic algorithm", IEEE International Conference on Intelligent Processing Systems, Beijing, China, 1997, pp. 137-142

[29] Z. Vasicek and L. Sekanina: "Formal Verification of Candidate Solutions for Post-Synthesis Evolutionary Optimization in Evolvable Hardware", Genetic Programming and Evolvable Machines, March 2011, pp. 1-23

[30] P. Fišer, J. Schmidt, Z. Vašíček, and L. Sekanina, "On Logic Synthesis of Conventionally Hard to Synthesize Circuits Using Genetic Programming", Proc. 13th IEEE Symposium on Design and Diagnostics of Electronic Systems, Vienna (Austria), 14.-16.4.2010, pp. 346-351.

[31] J. Hlavička and P. Fišer, "BOOM, a Heuristic Boolean Minimizer", Proc. International Conference on Computer-Aided Design, ICCAD 2001, San Jose, California (USA), 4.-8.11.2001, pp. 439-442

[32] P. Fišer and J. Hlavička, "BOOM - A Heuristic Boolean Minimizer", Computers and Informatics, Vol. 22, 2003, No. 1, pp. 19-51.

[33] P. Fišer, J. Hlavička, and H. Kubátová, "FC-Min: A Fast Multi-Output Boolean Minimizer", Proc. 29th Euromicro Symposium on Digital Systems Design (DSD'03), Antalya (TR), 1.-6.9.2003, pp. 451-454.

[34] P. Fišer and J. Schmidt, "It Is Better to Run Iterative Resynthesis on Parts of the Circuit", Proc. 19th of International Workshop on Logic and Synthesis 2010, Irvine, California (USA), 18.-20.6.2010, pp. 17-24.

[35] P. Fišer and J. Schmidt, "The Observed Role of Structure in Logic Synthesis Examples", Proc. 18th of International Workshop on Logic and Synthesis 2009, Berkeley, CA, USA, 31.7. - 2.8.2009, pp. 210-213.

[36] P. Fišer, J. Schmidt, "New Ways of Generating Large Realistic Benchmarks for Testing Synthesis Tools", Proc. 9th Int. Workshop on Boolean Problems, Freiberg, Germany, 16.-17.9.2010, pp. 157-164.

[37] J. A. Darringer et al., "Logic synthesis through local transformations", IBM Journal of Res. and Devel., vol. 25, no. 4, pp. 272-280, July, 1981.

[38] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", IEEE Trans. on Computer-Aided Design, Vol. 25(12), Dec. 2006, pp. 2894-2903.

[39] S. Yang, "Logic Synthesis and Optimization Benchmarks User Guide", Technical Report 1991-IWLS-UG-Saeyang, MCNC, Research Triangle Park, NC, January 1991.

[40] Berkeley Logic Interchange Format (BLIF), University of California, Brekeley, 2005.

[41] K. McElvain, "LGSynth93 Benchmark Set: Version 4.0", Mentor Graphics, May 1993.

[42] P. Fišer and J. Hlavička, "On the Use of Mutations in Boolean Minimization", Proc. Euromicro Symposium on Digital Systems Design (DSD'01), Warsaw (Poland), 4.-6.9.2001, pp. 300-305

[43] M. R. Garey and D. S. Johnson, Computers and Intractability: "A Guide to the Theory of NP-Completeness". San Francisco, CA: Freeman, 1979.

[44] F. A. Ogbu and D. K. Smith, "The application of the simulated annealing algorithm to the solution of *n/m/Cmax* flowshop problem". Computers & Ops. Res. 17, 243-253, 1990.

[45] A. J. Vakharia and Y-L. Chang, "A simulated annealing approach to scheduling a manufacturing cell." NRL, 37, 559-577, 1990.

[46] D. S. Johnson, C. R. Aragon, L. A. McGeoch, C. Scheron, "Optimization by simulated annealing: an experimental evaluation: part I, graph partitioning". Ops. Res., 37, 865-892, 1989