BRNO UNIVERSITY OF TECHNOLOGY

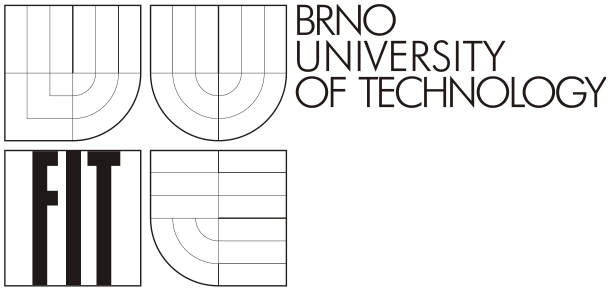
FACULTY OF INFORMATION TECHNOLOGY

RANDOMIZED ITERATIVE LOGIC SYNTHESIS ALGORITHMS

HABILITATION THESIS

AUTHOR ING. PETR FIŠER, PH.D.

BRNO 2012



FACULTY
OF INFORMATION
TECHNOLOGY

RANDOMIZED ITERATIVE LOGIC SYNTHESIS ALGORITHMS

HABILITATION THESIS

AUTHOR ING. PETR FIŠER, PH.D.

BRNO 2012

ABSTRACT

The logic synthesis is understood as a process of transforming a behavioral circuit description (typically register-transfer level – RTL) into a logic level description, typically a network of logic gates. By logic optimization we understand a process of refining this description to improve its quality, be it the size, delay, power consumption, etc.

However, these two terms (logic synthesis and optimization) are often used interchangeably or jointly in literature, and so they will be in this thesis.

Iteration at a *high-level* is proposed, as a more powerful, yet also more time-consuming alternative to the standard, single-pass synthesis and optimization process. The high-level attribute means that the iteration is performed over the whole synthesis process, not inside of one single synthesis step (algorithm).

Processes based on high-level iteration offer a possibility of reaching better results at expense of run-time. They are also adjustable in run-time: a trade-off between the result quality and run-time can be established by needs of the designer.

Iterative nature of the optimization algorithms brings a notion of state space. Therefore, logic optimization is viewed as a general combinatorial optimization problem in this thesis, in the sense of the state space concept. Notions of state space states and moves are introduced as valid optimization solutions and transformations from one to another, respectively.

Randomized versions of iterative algorithms offer higher iterative power. Locally optimum results are more easily avoided at expense of unpredictability of the final results, introduced by randomness.

Numerous iterative logic synthesis and optimization algorithms, even randomized, have been published in literature. They have different state spaces; some of them offer a global view of the problem. However, this work focuses only on several novel and original randomized iterative algorithms developed by the author. Their basic principles are described and emphasis is put on understanding the effect of both the randomness and iteration, and their degree necessary to make the algorithms perform well. The influence of randomness on the algorithm iterative power is studied as well.

Advantages and disadvantages of randomized iterative algorithms, compared to deterministic single-pass ones, are investigated. It will be shown that randomized iterative algorithms are capable of producing better results than deterministic single-pass ones, usually at expense of run-time.

The unquestionable merit offered by randomized iterative algorithms is a possibility of obtaining upper bounds of quality (e.g., of the size). These can be used, e.g., to evaluate the efficiency of other logic synthesis algorithms (benchmarking). It will be shown that some circuits can be simplified by orders of magnitude using iteration. If randomness is employed in addition, further simplification is possible.

Note that the well-known randomized iterative algorithms, the simulated annealing and evolutionary (genetic) algorithms are not studied here, since many studies on these topics were already published. These techniques were also tried to be applied to logic optimization, with various success. However, their properties and behavior are well known and, more importantly, can be generalized to any combinatorial optimization process. Hence, new algorithms developed just for purposes of logic synthesis and optimization are discussed here instead.

Also, sources of "external randomness" are explored. There are aspects that the synthesis process should not be influenced with. Ordering of variables or coordinate statements in the source file are such cases. However, this is not the case in practice. It will be shown that synthesis is crucially influenced by these; design tools produce very poor, or conversely, very good results under different orderings. Such kind of randomness is unknowingly introduced by, e.g., the designer, who naturally does not know the internals of the tools and definitely does not care about any, from his side meaningless reordering.

KEYWORDS

Logic synthesis, Boolean networks, multi-level optimization, two-level minimization, optimization, state space, iteration, randomness, iterative power.

ACKNOWLEDGEMENT

Most importantly, I would like to express many thanks to Jan Schmidt, for supporting my crazy ideas, for numerous fruitful discussions with him, for helping me see the direction, and for all his great ideas and his wisdom. Without him, I would probably get lost in a void of visions, without catching the relations and consequences.

Also many thanks to David Kovařík, for his hospitality, patience, and great Moravian wines he serves in his wine-bar. Without Jan and David, many ideas presented in this work would probably never emerge.

Last, but not least, I'd like to thank all the Californian IWLS workshop attendees, both from academia and industry. I thank them for not expelling me from the logic synthesis community because of my controversial research; my presentations there always brought amusement, irritation, and (most importantly) a great interest, all at the same time. I thank them for beneficial discussions, and for disclosing me their opinions and practical experiences. From this community, special thanks to Alan Mishchenko from UC Berkeley, for his enthusiasm for my research and for implementing several new commands in ABC that greatly helped me to conduct my research.

CONTENTS

1 INTRODUCTION

As the complexity of integrated circuits (ICs) progressively increases following the Moore's Law, bigger and bigger emphasis is put on modularity of designs. Complex ICs are constructed of smaller designs, like adders, multipliers, or custom cores. These are tailored together to form the final design, usually in a hierarchical way (ALUs contain adders, CPUs contain ALUs, systems on chip contain CPUs, etc.) The unquestionable advantage of such an approach is a possibility of design reuse; efficient implementations of frequently used small design features are known, or there are dedicated *generators* for them (like generic generators of adders). Then there is no need for optimization of the logic in the overall synthesis process. This makes synthesis of complex ICs fast and efficient.

However, the role of *random logic* is still pervasive. For example, controllers, arbiters, or other custom logic must be synthesized "from scratch", starting from their behavioral (e.g., RTL) description. This is where the *logic synthesis* and *optimization* plays the most important role. Note that the term *random* is used in a different context here – the logic is called random, since no regular patterns can be detected in its features arrangement (network).

Logic synthesis is usually understood as a process moving from a behavioral circuit description (RTL) to logic description (logic circuit – network of gates) [1]. By *logic optimization* we understand a process of finding a "better" representation of the same logic circuit, i.e., it operates at the same abstraction level. However, these two terms are often mixed up or used in connection, for simple reasons: for example, one may argue whether a truth table is a behavioral or gate-level description, or if an RTL code describing the circuit behavior using Boolean equations is still RTL [1], when truth table terms and equations, respectively, can be converted to gates in a straightforward way.

Even though logic synthesis and optimization is considered to be an already well-mastered and mature process, the research in this area still continues. These are some of the major driving forces:

1. Scalability. The volume of random logic progressively increases as well, thus the synthesis must be able to cope with increasingly larger designs.
2. Low-power designs. Since there is an increasing need for low-power, which can be achieved by, e.g. producing smaller designs, industry is keen to invest more resources to maximally decrease the design size. Low-power can also be achieved by special design techniques, which incorporate logic optimization too [2], [3].
3. Possibility of unexpectedly large results of high-level synthesis. Either the automated HDL synthesis itself may produce very bad results, or the HDL designer (HDL code programmer) may accidently describe the circuit in a bad way. Logic optimization should be able to simplify the network then. Ideally, logic optimization should produce (near-)optimum results independently of the initial description. It will be shown that reality is far from this ideal.
4. Design reuse. If a particular logic is reused many times in the design, it is highly required to optimize it as much as possible – an inefficiently synthesized small design feature may cause a size explosion of the complete design, if reused many times. Therefore, designers are keen to invest more effort to maximally optimize such components.

In many cases hardware designers may want to set a trade-off between the design quality (be it area, delay, power consumption, etc.) and design time. *Iterative* processes offer such a possibility. Particularly, the synthesis process may be iterated (repeated and refined by that), while the solution quality gradually increases in time. Then the iteration can be stopped judging by the designer's requirements (design quality, synthesis run-time).

If the whole synthesis process is iterated, we speak about *high-level iteration.*

Generally, iterative algorithms can be divided into two classes: *deterministic* and *randomized*. Deterministic algorithms typically rely on well elaborated control heuristics, while the randomized ones usually try to explore larger state space by introducing stochastic effects.

This work focuses on *randomized, high-level iterative algorithms* developed for purpose of logic synthesis and optimization. Several novel approaches to logic synthesis and optimization, both two-level and multi-level, are presented. Basically, two aspects are studied:

1. The possibility of trade-off between the solution quality and run-time. Particularly, we ask how the solution quality improves in time.
2. The effect of randomness. How does the randomness actually influence the synthesis? How much randomness is needed to reach satisfactory solutions? Partial derandomization is used to answer these questions.

1.1 LOGIC SYNTHESIS: SOME HISTORY AND STATE-OF-THE-ART

Basic principles of most of viable logic synthesis and optimization algorithms have been established already in early 1960's. Originally, the synthesis started with a two-level Sum-of-Products (SOP) description of the circuit or a truth table, i.e., a kind of behavioral description. The first algorithm minimizing SOP expressions was proposed by Quine and McCluskey [4], [5]. Then it was replaced by ESPRESSO [6], which became a well-established standard since 1980's.

After the SOP minimization, miscellaneous decomposition algorithms were applied to the result, to produce factored forms minimizing the number of literals [2], [3], [7], [8], [9]. After that the technology mapping process followed [2], [3].

In this "old book" approach, a high-level iteration was not possible, since transformation into the initial description (SOP) would completely destroy the obtained structure and thus waste all the effort.

All the referenced algorithms are based on SOP representations of functions (be it source descriptions or functions describing network nodes). Such a representation is not canonical and suffers from problems with scalability. Binary Decision Diagrams (BDDs) [10], [11] were introduced in 1980's, together with new, BDD-based algorithms, both for two-level [12], [13] and multi-level synthesis [14], [15]. However, even though BDDs represent logic functions implicitly, their size can easily blow up exponentially with the number of function's inputs as well [10], [11]. Therefore, only small BDDs are enforced in practice (so called *local BDDs* [15]), or they are used only for representing functions with a small number of inputs, e.g., for representing simple network nodes [16], [17], [18], [19], [20]. But definitely, BDDs are an ultimate solution when canonicity is required, or their special properties can be efficiently exploited by the synthesis algorithm [14], [15].

Many logic synthesis algorithms were implemented in academic tools MIS [16], SIS [17], and MVSIS [18] by Berkeley Logic Synthesis and Verification Group. In these tools, circuits are internally described as *Boolean networks*, whose nodes are represented as SOPs or BDDs.

Recently, the research shifted towards a different representation of networks: the *And-Inverter Graphs* (AIGs) [21], [22], [23], [24]. AIGs are more scalable and more uniform than standard tabular (truth table, SOP, PLA) circuit representations and new, more flexible synthesis and mapping algorithms may be applied upon these structures directly [24] - [29]. A synthesis tool ABC [19], [20] implementing these algorithms came as a successor of SIS and MVSIS and its development at UC Berkeley and other universities worldwide still continues.

ABC is presently the academic state-of-the-art. It is released as open-source software; new features can be easily implemented therein. Experiences obtained from ABC then reflect in industrial tools (EDA – Electronic Design Automation) development, since the authors of ABC closely collaborate with EDA industry.

Once a concept of *unified* circuit representation (network of SOPs, AIG) is introduced, both *resynthesis* and *high-level iteration* become possible. By resynthesis [29] we understand a single

synthesis process, where the forms of its input and output are the same, i.e., a process modifying the circuit in some way, while keeping the format of its description (AIG, for example).

High-level iteration means repeating the *whole synthesis process*, i.e., it can be understood also as a kind of resynthesis. The two necessary conditions for both (resynthesis and high-level iteration) are that optimization procedures must (1) operate with one representation only and (2) not destroy the circuit structure.

Synthesis in SIS is performed by executing several optimization steps, like node optimization by ESPRESSO [6], simplification using network don't cares [30], kernel and cube extraction [2], [3], [31], etc. The same happens in ABC, but most of the synthesis and optimization is performed upon AIGs. Algorithms like don't-care based node simplification [32], AIG rewriting [24], resubstitution, refactoring [25], [29], etc., are offered.

After that, technology mapping follows [2], [3], [26], [27], [33], [34], [35], usually into standard cells (ASIC technology library) or FPGA look-up tables (LUTs).

All these algorithms are implemented as individual *commands* in SIS and ABC.

As a consequence, plenty of synthesis, optimization, and technology mapping commands are available in these tools [17], [18], [19]. Unfortunately, it is impossible to determine a universal and ultimate sequence of these commands to be executed. Therefore, different *synthesis scripts* were proposed (e.g. "`script.rugged`" and "`script.algebraic`" in SIS, "`resyn`" scripts, "`choice`" and "`dch`" in ABC). These scripts are supposed to produce satisfactory, but definitely generally suboptimum results.

1.2 Logic Optimization and the State space Concept

Logic optimization, when viewed as a general combinatorial optimization problem [36], [37], is a search for a solution satisfying given constraints (functional equivalence with the origin) and optimizing the cost (quality), be it area, delay, power consumption, etc.

States in the state space represent different solutions; moves (operations) in the state space are transformations from one solution to another.

Most of the logic synthesis and optimization algorithms [2], [3], [6] are NP-hard [37], therefore the state space size grows exponentially with the instance size (be it the number of inputs, number of gates, signals). Using exact (optimum) algorithms is infeasible in practice, both due to the size of present circuits and the size of the state space induced. Therefore, simple greedy search algorithms are used [2], [3], [6], typically of the first-improvement or best-only nature [36]. The search is typically driven by some deterministic heuristics, hoping that the optimum (or at least near-optimum) solution will be obtained at the end.

Once the state space concept is introduced, we may ask what states are *reachable* by what algorithms (synthesis processes). This means, we ask what solutions are obtainable. It often happens that there are more different solutions of the same quality. So we ask: is it ever possible to obtain *all* of them by a given algorithm, under given circumstances? What's more, it could happen that the *optimum* solution cannot be reached by a given algorithm (synthesis process) at all. So we may ask whether a synthesis algorithm (search strategy) is *complete* in such sense [38]. Actually, by completeness we understand an *asymptotic convergence* to a globally optimum solution here; hence such an algorithm (state space search strategy) property will be denoted as *asymptotic completeness*.

Such a *completeness*, in sense of a *guarantee* of obtaining the optimum solution [38] in an *infinite time*, was e.g., proved for the simulated annealing algorithm [39], [40], [41], under specific circumstances.

Indeed, the completeness may be prevented by two aspects: the *state space* (optimum solutions are not present in the state space), and *control* (the algorithm needs not reach the optimum).

The notion of the state space and completeness of the logic optimization problem will be addressed specifically in the following two subsections and discussed in following sections, for particular algorithms.

1.3 ITERATIVE CIRCUIT OPTIMIZATION

The concept of iterative circuit optimization has been introduced in several different ways in the past. Most probably the first occurrence of iteration was in the rule-based optimization system LSS from IBM [42]. Here the logic network or the final mapped design indeed, is repeatedly refined by applying local transformations, i.e., substituting identified circuit patterns with different ones.

Another typical applications of iteration are algorithms based on simulated annealing (SA) [43], [44] and evolutionary processes (genetic algorithms, GA) [45], [46], [47]. Iteration is the conceptual basis of the algorithms.

By a *low-level iteration* we will understand a process, where the iteration is the basis of just a single synthesis step (e.g., iteration inside of the ABC "`dch`" command [19] or all the optimization processes mentioned in the paragraph above) or even its part only (e.g., the Kernighan-Lin partitioning procedure [48] in a technology mapping process).

On the other hand, authors of ABC suggest iterating the whole synthesis process [19], [20]. Particularly, the two phases – the technology independent optimization and technology mapping – are repeated several times, to improve the result quality. Structural hints obtained from the technology mapping can be further refined by re-running the technology independent optimization this way. Such an approach will be denoted as a *high-level iteration.*

When summarized, there may be several *levels* of iteration in the whole synthesis process, see examples in Figure 1. Single synthesis steps may contain low-level iterative algorithms, like SA, GA, Kernighan-Lin, etc. These steps, when combined, form the whole logic synthesis process that can be iterated (repeated) too, i.e., at a high level.

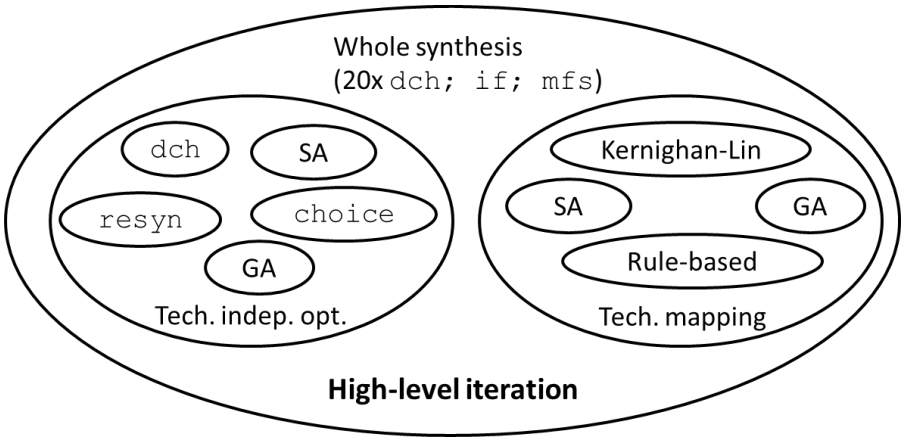


Figure 1. Levels of iteration

Essentially, any process that allows improving the result quality at expense of longer run-time will be considered as an *iterative circuit optimization*. Here the notion of *convergence* comes to importance. The solution quality should *improve* in time, not deteriorate. Next, the *stopping condition* should be defined. This is the point where the process is terminated and a result is returned. The stopping condition may be either a user-defined fixed number of iterations, timeout, the required result quality, or possibly an adaptive mechanism that needs not the user's intervention.

Two cases of iterative processes can be encountered, and both will be discussed in this thesis:

1. A complete and final circuit is gradually improved by iteration. The quality of the current result is known after completing each iteration, hence deciding on any chosen stopping condition is easy.
   From the state space point of view (see subsection 1.2), a new state is possibly reached after every iteration. Therefore, the more iterations are performed, the larger state space is explored.
2. Only bases (sources) for construction of the final solution are accumulated in course of iteration, whereas the solution is formed once, at the end of the process. This is the case of, e.g., BOOM (see Section 5).
   Here the concept of the state space is not as clear as in the previous point. More solutions are not generated by iteration. Instead, the *size* of the state space for the final, solution producing phase is gradually increased by iteration. Actually, since the optimization criterion cannot be computed after each iteration, we cannot consider the process as solving an iterative combinatorial optimization problem here.

1.4 RANDOMNESS

Because of high complexities of present designs, using exact (optimum) logic synthesis and optimization algorithms is not feasible. Therefore, approximate heuristic algorithms must be used in practice, as in ABC or SIS (see Subsection 1.1). Even though the employed heuristics usually produce solutions of sufficient quality, they do not guarantee optimum solutions and mostly do not even guarantee the maximum relative error.

Apart from algorithms presented in this work, many other randomized approaches to logic synthesis appeared. Simulated annealing [43], [44], evolutionary processes [45], [46], [47] are apparent cases, since randomness is essential there for success. However, these won't be discussed here, since their properties and behavior are already well known. Moreover, their application to logic synthesis is usually straightforward. *Dedicated* and newly developed logic synthesis algorithms will be studied instead.

All of the algorithms and logic synthesis systems (SIS, ABC) mentioned in Subsection 1.1 are fully deterministic; no random choices are made in the synthesis process. This brings a benefit of reproducibility of results – two runs of the algorithm using the same data produce equal results. However, the determinism may also involve inability of reaching different, possibly much better results, when the process is run repeatedly.

And what's more – even these deterministic algorithms mostly show hints of "unrecognized randomness". Particularly, the processes are usually greedy and they are not *systematic* [38]. Thus, some heuristic function is used to guide the search for the solution. Even though the heuristics are usually deterministic, there are often *multiple equally valued choices*. In such situations, the first occurrence is taken. Note that these choices are equally valued just at the point of decision and they will most likely influence the subsequent decisions. Therefore, different results *could* be produced, if different decisions were taken, without affecting the principles of the algorithms. In other words, the results obtained by one single deterministic algorithm heavily depend on its software implementation.

A typical example is traversing an AIG in a topological order, as e.g. in [24], [25]. The topological order is not unique, since there are usually more nodes in each topological level. Therefore, there remains some freedom in choosing the order which the nodes will be processed in. In ABC, nodes with the lowest ID (which is determined at the node creation instant) are processed first. Therefore, even the nodes creation order may influence the size and topology of the resulting AIG, which affects all the subsequently run processes.

Another example is the don't care based node simplification, as implemented in SIS [17], [32]. Again, observability don't cares are extracted by topological traversal of the network. By introducing different ordering at each topological level, different results can be obtained.

Taking this into account, a simple way of randomization of deterministic processes is offered. However, this won't be the topic of this study.

From the state space point of view, deterministic algorithms seem to traverse only a very limited portion of it. In an ideal case, they will find a way to the global optimum or to a solution likely near to optimum, due to efficient pruning heuristics [6]. However, this needs not the practice, as some results indicate [49], [50]. To solve this problem, we need either very sophisticated heuristics (which is not likely), or to explore larger state space. Therefore, some kind of *diversification* could help. And this can easily be accomplished by introducing *randomness*.

One of the first attempts to use a randomized algorithm in logic synthesis in this sense was proposed in [51]. The core optimization algorithm is deterministic; however it is suggested to be run repeatedly, with random initial starting points. Getting stuck in a local optimum is avoided this way and a larger space of solutions is explored.

1.5 RANDOMIZED ITERATIVE ALGORITHMS

The concepts of iteration and randomness, when combined, form a special class of logic synthesis and optimization algorithms. Simulated annealing [43], [44] or evolutionary processes [45], [46], [47] are apparent state-of-the-art cases, since both randomness and iteration are involved here, from the very nature of the processes. Several other algorithms, where randomness and iteration is exploited too, will be introduced and studied more thoroughly in the following sections.

Up to the knowledge of the author, there was no research on randomness introduced at a high level, i.e., randomness introduced "from outside", in order to influence the whole synthesis process, not only optimization. This will be the case of two of the presented processes. Iteration at a high level was not thoroughly studied as well.

A common property of randomized iterative algorithms is the possibility of trade-off between the run-time and solution quality. Moreover, coming from their random nature, the optimum solution can be obtained in an infinite time, provided that the algorithms are designed properly (they are asymptotically complete, see Subsection 1.2).

This is, of course, just a theoretical conclusion without any practical impact. However, arbitrarily precise *upper bounds* (e.g., of size) can be obtained by randomized iterative algorithms. These can be used, e.g., for testing the efficiency of other logic synthesis algorithms (benchmarking), or to obtain estimations of a theoretical circuit complexity [52], [53], [54], [55].

Generally, randomness introduced into iteration helps the algorithm to keep the convergence longer. Therefore, better results can be obtained, compared to deterministic iterative algorithms.

1.6 ACCEPTANCE OF RANDOMIZED ITERATIVE ALGORITHMS

As it was stated above, iterative synthesis allows the designer to set a trade-off between the solution quality and design time. However, it is still not very well accepted by EDA industry (the producers of the algorithms) [56]. Commercial tools are designed preferably for speed, since their run-time is the bottleneck in the large ICs design. Nevertheless, if a significant quality improvement were offered by iteration, it could be accepted. Current results do show such a potential.

Conversely enough, iteration *would be* probably accepted by ICs design industry (the consumers of the algorithms), if offered by EDA vendors [57]. Iterative processes would be most likely used for low-power, low-area, and very high speed designs, where designers struggle to improve the target quality criterion (power consumption, area, delay) by units of per-cents or they try to fit into prescribed constraints.

Last, but not least, one application is in the design of small pieces of random logic, that are to be frequently reused. These should be synthesized as precisely as possible, since even small size (delay) differences may cause big differences in the final design.

Unfortunately enough, randomness is accepted by the EDA industry with the same disgust as iteration [56]. Reproducibility of the results is one of the essential issues. However, we do not speak about *true randomness* – it is just *pseudo-randomness*. Therefore, the reproducibility can be easily ensured by, e.g., fixing the pseudo-random generator seeds, without losing any of the mentioned benefits. Moreover, the seed can also be specified as one of the synthesis parameters. Then, a possibility of obtaining different solutions will emerge, without losing the benefits of reproducibility.

Then again, consumers of the EDA software (IC designers) need not be affected by introduced randomness at all. Actually, they could even welcome it, as a novel possibility of generating structurally different designs, i.e., several options they can choose from [57].

2 INFLUENCE OF THE SOURCE FILE STRUCTURE

It can be observed that many synthesis processes are not immune to the structure of the source file, like the ordering of variables [58], [59] and ordering and syntax of HDL statements [60]. Therefore, different runs of one process with differently structured source file produce different results. Possible reasons for it will be discussed in this Section and some quantitative results will be given.

Typically, in the synthesis algorithms variables are processed in a lexicographical order, which is defined a-priori, usually by their order in the source file. Then, different orderings of variables may make heuristic algorithms run differently, possibly producing different (but definitely still correct) results.

A typical and well known example of such a behavior are BDDs [10], [11]. Here the ordering of variables is essential; the BDD size may explode exponentially with the number of variables under a "bad" ordering [10]. Computing the optimum ordering of variables is NP-hard itself [61], thus infeasible in practice. Even though there are efficient heuristics for determining a possibly good variable ordering [62], they consume some time, whereas do not guarantee any success, and thus they are usually not employed in practice. Typically, the default variable ordering in the BDD manipulation package CUDD [63] (which is used in SIS and ABC too) is *just equal* to the ordering of variables in the source file – no reordering technique is employed.

Another, and more important example, is the topological traversal of AIG nodes in algorithms employed in ABC and SIS (see Subsection 1.4). Even different orderings of input and output variables will involve different AIGs or differently arranged networks (in sense of their internal representation).

Also the well-known two-level Boolean minimizer ESPRESSO [6] (which is used both in SIS and ABC too) is sensitive to ordering of variables. There are many essential parts of the overall algorithm, where decisions are made in a lexicographical way. Some decisions do not influence the result quality; they just may influence the run-time (e.g., in the tautology checking process), some do influence the result as well (e.g., the Irredundant phase) [6].

Therefore, even changing the variables ordering in the source file header (be it PLA [6] for ESPRESSO or BLIF [64] for ABC) can significantly affect the algorithms runs and induce different results. It will be documented in the following Subsection how serious differences there are in practice.

Also, some commercial synthesis tools are sensitive even to the order of nodes (which are coordinate RTL statements). This issue will be documented here as well. For an experimental study of the influence of small modifications of the RTL code on the result, see [60].

2.1 EXPERIMENTAL RESULTS

The experimental evaluation of several basic optimization and technology mapping commands in ABC [19], technology independent optimization scripts (which comprise of the basic synthesis commands), and complete synthesis scripts, targeted to standard cells (the "`strash; dch; map`" script) and look-up tables (4-LUTs), the "`strash; dch; if; mfs`" script, will be presented here. Finally, results of ESPRESSO [6] and even ESPRESSO-EXACT are shown. The dependency on both the input and output variables ordering will be studied.

No influence of the PLA terms ordering or nodes ordering in BLIF as observed in ESPRESSO or any of the studied processes in ABC.

The ABC experiments were conducted as follows: 228 benchmarks from the IWLS and LGSynth benchmarks sets [65], [66] were processed. Given a benchmark, its inputs and/or outputs were randomly permuted in the source BLIF file [64] (or PLA for ESPRESSO), the synthesis command was executed, and the number of AIG nodes, gates, LUTs or literals, respectively, was measured. This was repeated 1,000-times for each circuit.

In order to compactly represent all the results, the maximum and average percentages of size differences (minimum vs. maximum) were computed, over all the 228 circuits. The results are shown in Table 1.

We can observe striking size differences (up to more than 95%), especially for the complete synthesis processes. Even the numbers of literals obtained by ESPRESSO-EXACT differ, since ESPRESSO-EXACT guarantees minimality of the number of terms only, nothing is guaranteed for literals.

Table 1. Influence of permutation of variables – summary results (1 iteration)

	Process	Unit	Permuted inputs		Permuted outputs		Permuted both	
			max.	*avg.*	*max.*	*avg.*	*max.*	*avg.*
Technology independent optimization: commands	`balance`	AIG	7.69%	1.04%	11.48%	1.60%	12.50%	2.27%
	`rewrite`	AIG	15.38%	0.68%	19.30%	2.41%	19.13%	2.78%
	`refactor`	AIG	12.07%	0.36%	29.73%	2.49%	29.73%	2.79%
	`resub`	AIG	2.50%	0.06%	20.83%	1.70%	20.83%	1.71%
Technology independent optimization: scripts	`resyn2`	AIG	44.53%	4.60%	52.75%	5.58%	52.69%	7.38%
	`resyn3`	AIG	13.56%	1.57%	22.50%	2.74%	22.66%	3.72%
	`choice`	AIG	34.40%	7.17%	38.14%	7.14%	36.17%	10.13%
	`dch`	AIG	60.53%	10.42%	40.39%	9.33%	60.50%	13.50%
Technology mapping	`map`	gates	17.09%	1.35%	12.28%	1.93%	17.09%	2.84%
	`fpga`	LUTs	0.00%	0.00%	5.26%	0.29%	5.26%	0.29%
	`if`	LUTs	0.00%	0.00%	2.88%	0.24%	2.88%	0.24%
Complete synthesis	`strash; dch; map`	gates	74.38%	8.67%	70.47%	10.52%	86.27%	13.40%
	`strash; dch; if; mfs`	LUTs	92.14%	11.50%	85.42%	12.60%	95.07%	14.81%
Two-level optimization	ESPRESSO	literals	34.90%	1.51%	11.82%	1.04%	42.95%	2.11%
	ESPRESSO-EXACT	literals	0.63%	0.02%	6.06%	0.23%	6.06%	0.24%

Next, detailed results for two particular circuits, *apex2* and *cordic* [65] are shown in Table 2 and Table 3. For each process, the minimum, maximum, and average values are presented, together with percentage differences between the minima and maxima. More precise results were computed here; they were obtained from 10,000 runs. ESPRESSO is insensitive to ordering of variables for these particular circuits, thus the results are not present.

When observing the results of the individual synthesis processes and the overall synthesis, the behavior of the *apex2* case is expectable. Almost all the synthesis processes were sensitive to the ordering of variables, and the effect accumulates in the progress.

However, *cordic* is quite a striking example. This is incidentally the circuit responsible for the maximum difference of LUTs counts in the complete synthesis process "`strash; dch; if; mfs`" in Table 1. Solutions ranging from 27 to 687 LUTs were obtained. But, strangely enough, the standalone synthesis processes ("`strash`", "`dch`", "`if`", "`mfs`") are *not significantly sensitive* to variables ordering (e.g., the mapping phase is completely immune). In quantitative measures, the effects of individual processes can never be combined to obtain such differences in the final design size. Therefore, we must conclude that some *qualitative* flaws occur in the progress. There are hints that structural choices [67], [68] are responsible for this phenomenon.

Table 2. Influence of permutation of variables – details for *apex2*

	Process	Unit	Permuted inputs				Permuted outputs				Permuted both			
			min.	*max.*	*avg.*	*%*	*min.*	*max.*	*avg.*	*%*	*min.*	*max.*	*avg.*	*%*
Technology independent optimization: commands	`balance`	AIG	4162	4191	4174.2	0.69%	4155	4180	4170.6	0.60%	4150	4202	4176.3	1.24%
	`rewrite`	AIG	4129	4137	4132.7	0.19%	4132	4138	4134.8	0.14%	4128	4139	4133.4	0.27%
	`refactor`	AIG	4018	4018	4018.0	0.00%	4018	4027	4022.9	0.22%	4018	4027	4022.8	0.22%
	`resub`	AIG	4302	4317	4309.6	0.35%	4301	4308	4304.4	0.16%	4300	4322	4311.6	0.51%
Technology independent optimization: scripts	`resyn2`	AIG	3360	3448	3399.9	2.55%	3389	3422	3407.4	0.96%	3351	3450	3403.3	2.87%
	`resyn3`	AIG	3918	3945	3927.3	0.68%	3874	3930	3909.8	1.42%	3859	3948	3909.8	2.25%
	`choice`	AIG	4419	4522	4494.0	2.28%	4490	4508	4499.0	0.40%	4419	4524	4492.8	2.32%
	`dch`	AIG	2931	3194	3072.3	8.23%	3008	3143	3067.3	4.30%	2918	3198	3063.5	8.76%
Technology mapping	`map`	gates	4371	4401	4383.7	0.68%	4354	4383	4371.5	0.66%	4350	4402	4380.1	1.18%
	`fpga`	LUTs	2013	2030	2020.1	0.84%	2014	2020	2017.5	0.30%	2006	2029	2016.4	1.13%
	`if`	LUTs	2040	2040	2040.0	0.00%	2039	2040	2039.5	0.05%	2039	2040	2039.5	0.05%
Complete synthesis	`strash; dch; map`	gates	3221	3552	3378.7	9.32%	3292	3464	3360.5	4.97%	3202	3559	3369.8	10.03%
	`strash; dch; if; mfs`	LUTs	1502	1731	1631.0	13.23%	1587	1666	1628.3	4.74%	1508	1744	1631.2	13.53%

Table 3. Influence of permutation of variables – details for *cordic*

	Process	Unit	Permuted inputs				Permuted outputs				Permuted both			
			min.	*max.*	*avg.*	*%*	*min.*	*max.*	*avg.*	*%*	*min.*	*max.*	*avg.*	*%*
Technology independent optimization: commands	`balance`	AIG	2727	2735	2730.7	0.29%	2727	2728	2727.5	0.04%	2727	2735	2730.5	0.29%
	`rewrite`	AIG	989	991	990.0	0.20%	987	991	989.0	0.40%	987	991	988.9	0.40%
	`refactor`	AIG	1125	1129	1127.0	0.35%	1128	1128	1128.0	0.00%	1125	1129	1127.0	0.35%
	`resub`	AIG	2723	2723	2723.0	0.00%	2723	2723	2723.0	0.00%	2723	2723	2723.0	0.00%
Technology independent optimization: scripts	`resyn2`	AIG	463	537	502.5	13.78%	487	492	489.5	1.02%	459	541	502.3	15.16%
	`resyn3`	AIG	2677	2724	2695.0	1.73%	2685	2685	2685.0	0.00%	2677	2724	2696.0	1.73%
	`choice`	AIG	2440	2773	2764.0	12.01%	2770	2770	2770.0	0.00%	2440	2774	2761.8	12.04%
	`dch`	AIG	396	545	486.3	27.34%	448	518	482.8	13.51%	411	555	490.7	25.95%
Technology mapping	`map`	gates	2762	2772	2766.7	0.36%	2765	2766	2765.5	0.04%	2761	2772	2766.2	0.40%
	`fpga`	LUTs	930	932	931.0	0.21%	931	931	931.0	0.00%	930	932	931.0	0.21%
	`if`	LUTs	804	804	804.0	0.00%	804	804	804.0	0.00%	804	804	804.0	0.00%
Complete synthesis	`strash; dch; map`	gates	447	2409	567.1	81.44%	486	597	541.2	18.59%	460	2412	571.2	80.93%
	`strash; dch; if; mfs`	LUTs	27	687	335.5	96.07%	178	676	425.5	73.67%	34	689	318.4	95.07%

Distributions of frequencies of occurrence of solutions of a given size are shown in Figure 2 and Figure 3, for the *apex2* and *cordic* circuits. The complete 4-LUT synthesis script ("`strash; dch; if; mfs`") was executed, for 100,000 different orderings of variables. The result obtained using the original ordering is indicated by the bold vertical line.

We can see a Gaussian-like distribution for the *apex2* circuit, or actually, a superposition of two Gaussian distributions. Even the original ordering of variables falls to the "better" part of the chart.

For the *cordic* circuit we can observe two completely isolated regions. There are apparently two or more classes of similar implementations (similar in size, probably similar in structure too), which synthesis produce depending on the ordering of variables. This phenomenon is still under examination, reasons for it are discussable. Note that the *apex2* case also shows hints of two structurally different classes of solutions.

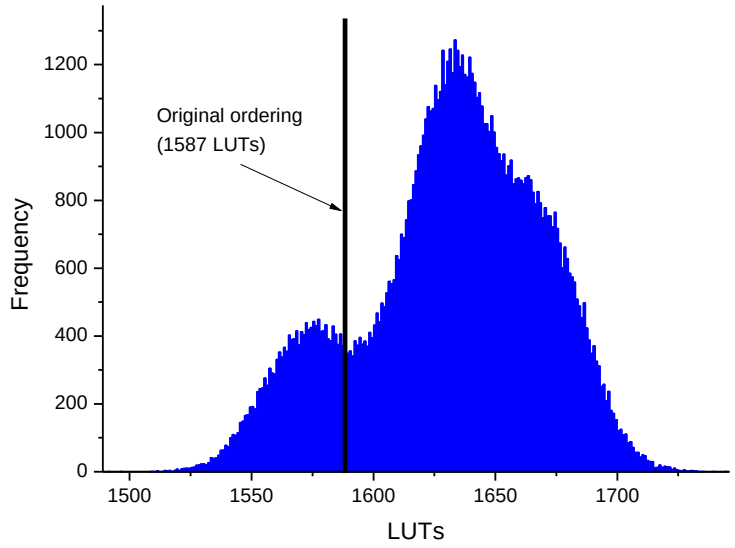


Figure 2. Distribution of solutions – *apex2, 1 iteration*

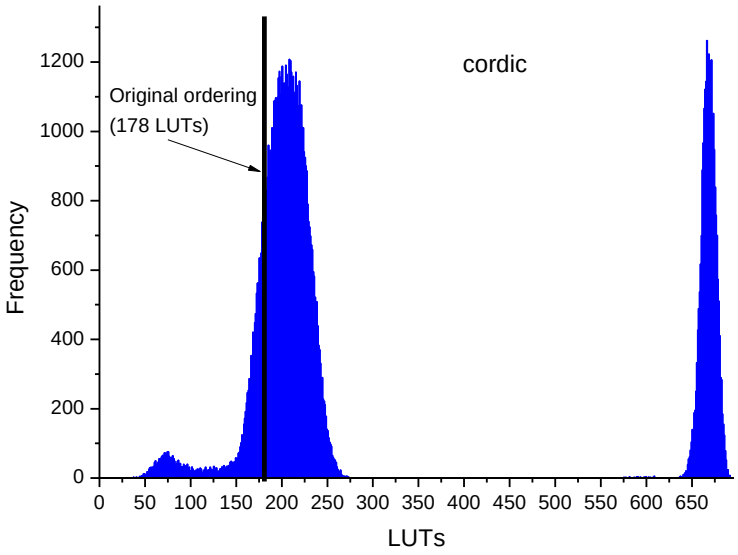


Figure 3. Distribution of solutions – *cordic, 1 iteration*

Since it is suggested to run the ABC synthesis scripts (or even the individual commands) several times to improve the result quality, several selected synthesis processes were run in an iterative way, for 20 iterations, to see if the striking size differences shown in Table 1 were caused just by a "bad luck" and if the iterative process will converge to a stable solution, independently of the ordering. The summary results are shown in Table 4.

We see that even though some peak values are slightly reduced (e.g., the "`strash; dch; if; mfs`" process), some processes become even more sensitive to ordering of variables (e.g., "`refactor`"). Generally, the average sensitivity increases by iteration.

Table 4. Influence of permutation of variables – summary results (20 iterations)

	Process	Unit	Permuted inputs		Permuted outputs		Permuted both	
			max.	*avg.*	*max.*	*avg.*	*max.*	*avg.*
Technology independent optimization: commands	`balance`	AIG	18.75%	1.37%	10.74%	1.43%	18.75%	2.44%
	`rewrite`	AIG	9.62%	0.91%	26.61%	2.01%	26.61%	2.54%
	`refactor`	AIG	69.52%	0.97%	62.57%	2.40%	70.14%	2.87%
	`resub`	AIG	1.54%	0.04%	15.00%	1.24%	15.00%	1.25%
Technology independent optimization: scripts	`resyn2`	AIG	73.96%	7.92%	74.06%	7.42%	78.45%	10.21%
	`resyn3`	AIG	18.75%	1.84%	34.54%	2.76%	40.91%	3.90%
	`dch`	AIG	82.82%	20.49%	67.19%	18.75%	81.07%	23.72%
Technology mapping	`fpga`	LUTs	52.94%	10.34%	44.00%	6.90%	56.60%	12.00%
	`if`	LUTs	2.25%	0.04%	2.88%	0.25%	2.88%	0.25%
Complete synthesis	`strash;dch; if; mfs`	LUTs	88.89%	18.84%	78.71%	15.77%	85.71%	20.01%

Distributions of frequencies of the obtained solutions for the *apex2* and *cordic* circuits, where the LUT-mapping process was run for 20 iterations, are shown in Figure 4 and Figure 5, respectively. Again, 100,000 different orderings of variables were tried.

We see that the result quality was significantly improved in both cases and has more steady distribution, shifted towards more positive values. The area, where *cordic* was synthesized very poorly, completely disappeared. But still, the original ordering of variables yields statistically very inferior results.

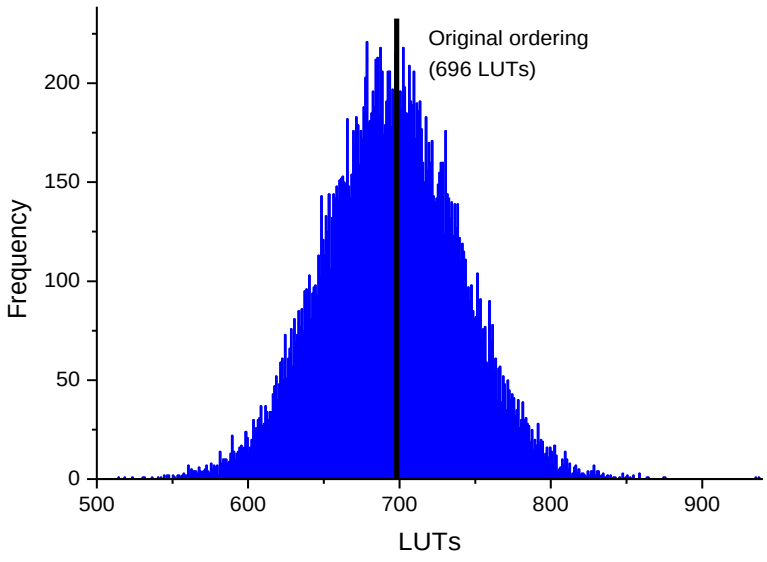


Figure 4. Distribution of solutions – *apex2, 20 iterations*

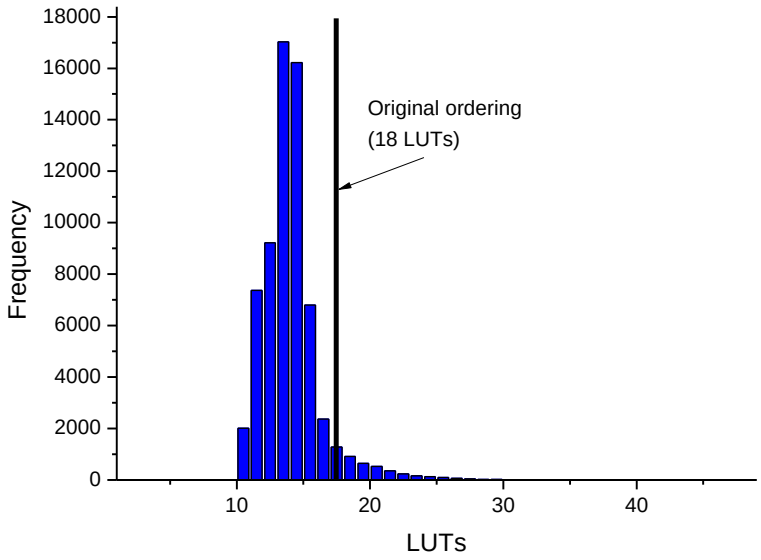


Figure 5. Distribution of solutions – *cordic, 20 iterations*

2.2 COMMERCIAL TOOLS

Dependency of the result quality on the ordering of variables was observed in commercial tools too. Two tools were studied and both were found to be very sensitive to the structure of the HDL statements. Surprisingly enough, the tools were also sensitive to a mere reordering of the gates instantiation, i.e., coordinate statements in the HDL code, which was not the case of any examined process in ABC.

The experiment started with BLIF [64] descriptions and after permuting the variables (and nodes in the BLIF file), each benchmark was converted to VHDL and processed by commercial LUT-mapping synthesis. The numbers of 4-LUTs in the results was measured. Summary results of the 228 benchmarks [65], [66] are shown in Table 5. Again, maximum and average differences in the obtained LUT counts are shown.

Table 5. Influence of permutation of variables and nodes – commercial tools

Tool	Permuted inputs		Permuted outputs		Permuted nodes		Permuted all	
	max.	*avg.*	*max.*	*avg.*	*max.*	*avg.*	*max.*	*avg.*
#1	0.00%	0.00%	0.00%	0.00%	15.76%	0.21%	17.26%	0.22%
#2	43.62%	4.71%	52.19%	5.57%	38.81%	3.40%	66.62%	9.23%

2.3 SOLUTIONS ANALYSIS – STRUCTURAL DIFFERENCE

It was shown that solutions of different sizes can be obtained by permuting the variables. This fact is documented in the four above histograms well enough. However, one may ask whether frequent occurrences of one particular solution size represent only a single, frequently produced solution, or there are many *structurally different* solutions of equal size. Surprisingly, the latter is the case. This is documented by a histogram in Figure 6, for the *cordic* circuit, 20 iterations of the LUT synthesis process ("`strash; dch; if; mfs`"). Numbers of both *all* (the complete bars) and *structurally different* (solid bars) solutions were recorded. Actually, the complete bars in Figure 6 correspond to Figure 5, only less orderings were exercised (10,000), due to the time overhead caused by the structural equivalence checking process. As a result we see that the shape of the histogram of structurally different solutions is completely the same as the one of all solutions.

To state some concrete data for this experiment: out of 10,000 random permutations, there were 78 structurally different optimum 10-LUT solutions found and 7,616 different solutions in total (which is 76%). Therefore, the potential of obtaining different solutions is very high.

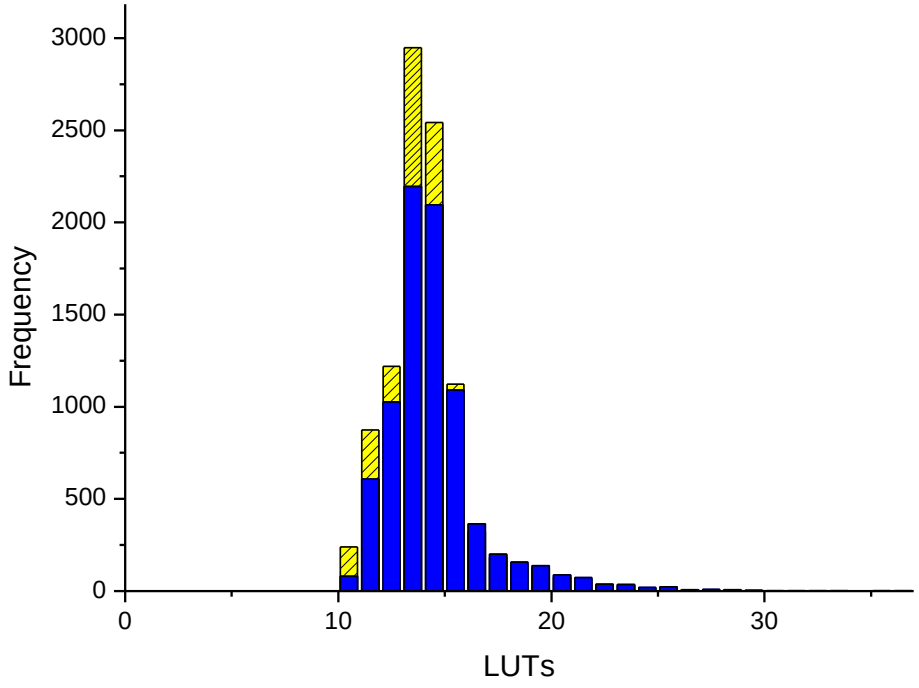


Figure 6. Distributions of different solutions – *cordic, 20 iterations*

2.4 DISCUSSION

We have seen that most of basic synthesis and optimization algorithms are not immune to ordering of variables and statements in the source circuit description; the source file structure sometimes significantly influences the synthesis result. One of the possible reasons for it is the *lexicographical* processing of variables. This means, a particular ordering of variables is introduced just by reading the input and construction of internal structures.

Let us note that, for the algorithms, any ordering of variables is as random as any other. Therefore, a kind of randomness is introduced just by the very circuit specification, that may arrive, e.g., from the RTL synthesis in practice. Also, the designer (HDL programmer) himself introduces such randomness. We have seen striking examples, where the synthesized circuit size was reduced up to 25-times just by reordering of variables. Even iteration will not help too much; generally it even increases the difference between minimum and maximum result sizes.

Effects caused by different orderings of variables in BDDs [10], [11] will not be studied here, since the conclusions would be apparent – it is a well-known fact that the BDD size may explode exponentially with different variables ordering [10]. This is the reason why the contemporary synthesis algorithms try to avoid BDDs completely, or to go down to so called *local BDDs* [15], where the size explosion is prevented.

Actually, ABC uses global BDDs only rarely (e.g. for circuit collapsing and disjoint-support decomposition [69]), or only in cases where the size explosion cannot happen because of the limited number of inputs [70]. Therefore, BDDs are definitely not responsible for the result size differences presented in this section.

Sensitivity to the ordering of nodes (gate instantiations) was not observed in ABC, however, two studied commercial tools were sensitive to it. Any reasoning about this fact would be speculation, therefore it will be left upon fantasy of the readers.

Now we still may ask two ultimate questions:

"What will happen, if I just reorder the variables in the source file header (ports and signals definition) or reorder the statements?" and

"What shall happen, if I just reorder the variables in the source file header (ports and signals definition) or reorder the statements?"

But next, we may also think about exploiting these facts to *systematically improve* logic synthesis. The following section will be devoted to this idea.

3 IMPROVING THE ITERATIVE POWER BY PERMUTATION

The state-of-the-art high-level iterative process, as it can be used, e.g., in ABC, can be described as follows: first, an internal representation (SOP, AIG, network of gates, network of BDDs, etc.) for the technology independent optimization is generated from the initial description or the mapped network (e.g., from the BLIF file [64]). Then a technology independent optimization, followed by technology mapping is performed. The process is repeated (iterated), until the stopping condition (number of iterations, result quality, timeout, etc.) is satisfied, see Figure 7.

The general aim of the process is to transform the initial circuit description into the target technology (ASIC library gates, FPGA LUTs), while trying to optimize the quality (size, delay, power consumption) of the solution.

```
do {
     generate_internal_representation
     technology_independent_optimization
     technology_mapping
} while (!stop)
```

Figure 7. The iterative resynthesis

Assuming that each iteration does not deteriorate the solution, the solution quality improves in time. This needs not be true in practice, however. For such cases several options are possible:

1. to hope that the overall process will "recover" from small deteriorations,
2. to accept only improving (non-deteriorating) iterations,
3. to record the best solution ever obtained and return it as the final result,
4. combination of 1) and 3).

The first and the last options are usually used in practice.

Usually it happens that the iterative process quickly converges to a stable solution, which does not improve any more in time. In an ideal case it is the best possible solution (global optimum). However, usually this is not the case in practice; such an iterative process tends to get stuck in a local optimum [47], [59].

Just a slight modification of the algorithm from Figure 7 might help to escape local optima and thus improve the *iterative power* of the resynthesis [59]:

```
do {
     randomly_permute_variables
     generate_internal_representation
     technology_independent_optimization
     technology_mapping
} while (!stop)
```

Figure 8. The iterative resynthesis with random permutations

Here only the `randomly_permute_variables` step was added, where the random reordering of variables (input, output, or both) is performed. This step can be executed in a time linear with the number of variables, hence it does not bring any significant time overhead.

Note that, unlike in the previous section, the reordering of variables is performed in each iteration, not only at the beginning of the iterative synthesis process. Therefore, the permutations effects may accumulate.

3.1 EXPERIMENTAL RESULTS

Very exhaustive experiments were performed in order to justify the benefit of using random permutation of variables in the high-level iteration process. There were processed 490 benchmark circuits, coming from academic IWLS and LGSynth benchmark suites [65], [66], as well as from large industrial designs from OpenCores [71] (having up to 100,000 LUTs after synthesis). The 4-LUTs mapping process was chosen for testing purposes. However, the same behavior can be expected for any target technology.

The most recent LUT-mapping synthesis script suggested by the authors of ABC was used: "`strash; dch; if; mfs; print_stats -b`" as a reference. Then, the ABC command "`permute`" randomly permuting both inputs and outputs was implemented and employed, yielding the script "`permute; strash; dch; if; mfs; print_stats -b`". Both scripts were executed 20-, 100-, 1,000-, and 5,000-times for each circuit, while the best result ever obtained was recorded and returned as the solution (this is accomplished by the "`print_stats -b`" command). The numbers of resulting 4-LUTs and the delay (in terms of the length of the longest path – the circuit levels) were measured.

Results for all the 490 circuits are shown in Figure 9 and Figure 10, for area (4-LUTs) and delay (levels), respectively. The scatter-graphs visualize the relative improvements w.r.t. no permutations used. Positive values indicate an improvement, the negative ones deterioration. The size of the original mapped circuit, in terms of 4-LUTs, is indicated on the x-axis. Two border cases, 20 and 5,000 iterations are shown here only. Results of 100 and 1,000 iterations lay in-between.

We see that a significant improvement may be reached even when the process is run for 20 iterations. However, also more deteriorating cases are observed. When iterated more, the results become more positive, especially for larger circuits. This is quite obvious, since these circuits usually converge slower (see Subsection 3.2).

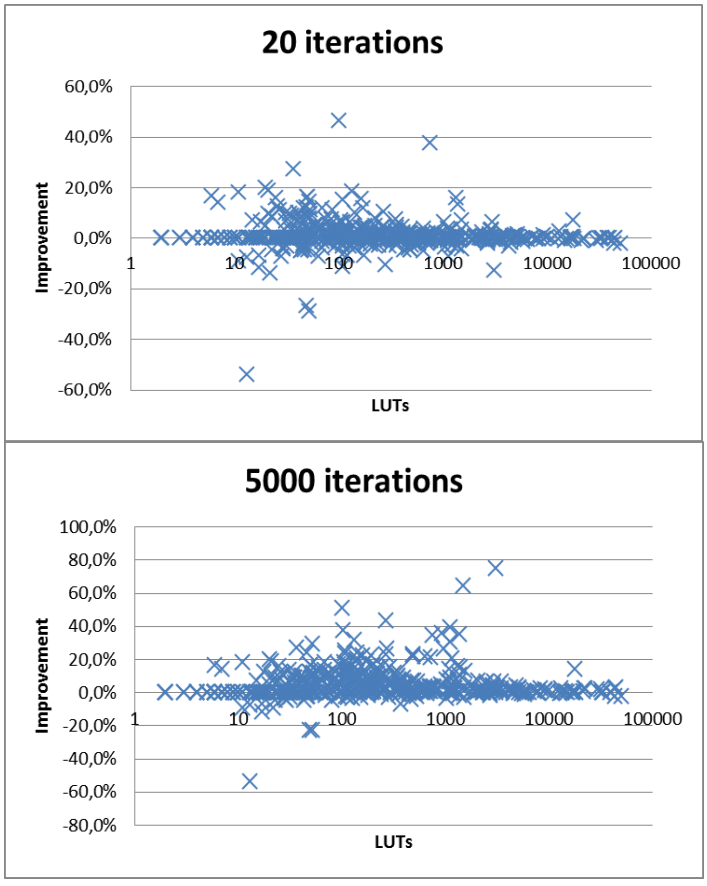


Figure 9. Area improvements w.r.t the standard iterative process

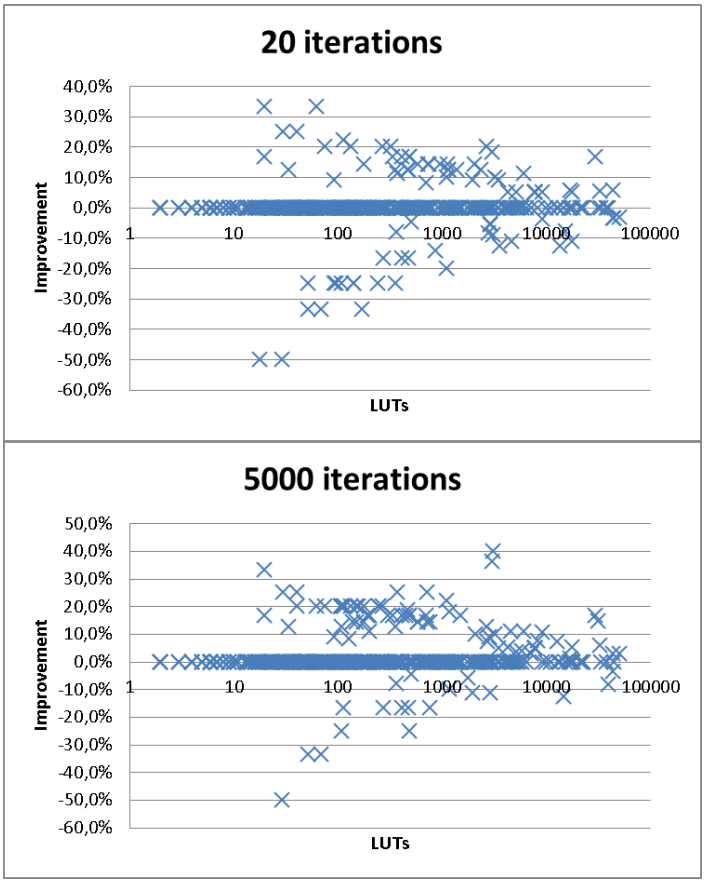


Figure 10. Delay improvements w.r.t the standard iterative process

Summary statistics are shown in Table 6. Only 290 circuits, whose resulting implementation exceeded 100 LUTs, were accounted in these statistics, to make the practical impact more credible. The minimum, maximum and average percentage improvements for both area and delay are given. Also the percentages of cases, where the improvement is positive ("*Better in*") and negative ("*Worse in*"), are shown. The complement to 100% of the sum of these two values represents cases where solutions of equal quality (LUTs, levels) were obtained.

Table 6. Summary statistics

Iterations		20	100	1,000	5,000
LUTs	*Minimum*	-12.8%	-8.2%	-5.4%	-6.7%
	Maximum	46.5%	51.2%	74.6%	75.2%
	Average	1.0%	2.1%	4.9%	6.1%
	Better in	52.2%	64.9%	81.0%	82.6%
	Worse in	39.8%	28.8%	15.2%	13.9%
Levels	*Minimum*	-33.3%	-33.3%	-25.0%	-25.0%
	Maximum	22.2%	27.3%	40.0%	40.0%
	Average	0.6%	0.6%	1.6%	2.5%
	Better in	16.3%	13.8%	19.7%	23.9%
	Worse in	9.3%	5.5%	6.2%	5.5%

We see that with an increasing number of iterations the results become more stable and tend to improve, both in area and delay. There is a positive average improvement obtained even for 20 iterations run. For the 5,000 iterations case the average improvement reaches 6.1% in area and 2.5% in delay. Also cases, where deterioration was obtained, are becoming rare (13.9% and 5.5% for area and delay, respectively).

Let us make a theoretical reasoning about the observed facts now. Assume the worst case, where the number of deteriorating (w.r.t. the process with no permutations used) solutions of one iteration of resynthesis is 50% (equal chance for both the improvement and deterioration). Then, also chances for improvement of the overall process would be 50%. However, in Table 6 we see that all the minimum improvements (maximum deteriorations) are much less than 50%, even for 20 iterations. From these figures we can conclude that permutation always pays off.

3.2 THE CONVERGENCE ANALYSIS

Illustrative examples of convergence curves for the iterative synthesis with and without using random permutations for two of the LGSynth benchmark circuits [66] *alu4* and *apex2* are shown in Figure 11. The progress of the size reduction during 1,000 iterations was traced.

Here we see an experimental justification of the presented theory. In general, it is not possible to say what method converges faster. Theoretically, both should converge equally fast. This can be seen, e.g., in the *alu4* case, where the standard synthesis converges faster at the beginning, but then the convergence slows down. When the resynthesis without using permutations converges to a local minimum, the permutations will help to escape it (see the *apex2* curves – here the local minimum was reached around the 300th iteration, whereas the solution quality still improves after 1,000 iterations when permutations are used). Similar behavior can be observed for most of the tested circuits. This confirms the theory – the *permutations do increase* the iterative power and helps to keep the convergence longer.

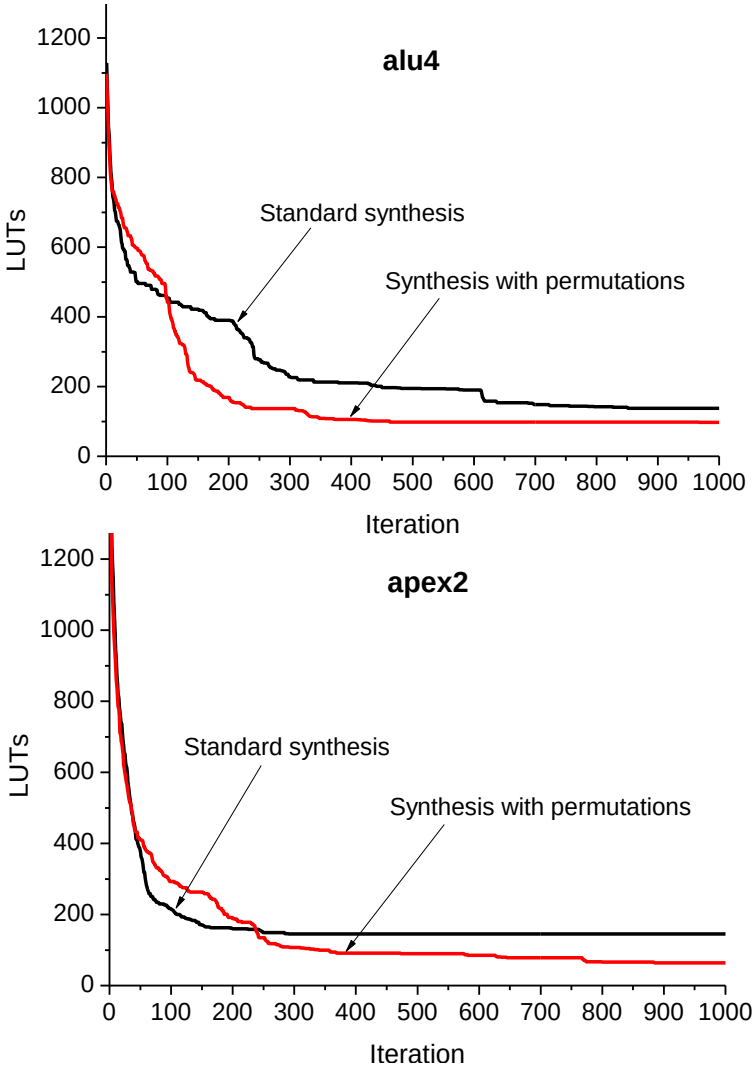


Figure 11. Convergence curves for the *alu4* and *apex2* circuits

3.3 Asymptotic Completeness of the Algorithm

The notion of asymptotic completeness of iterative logic optimization algorithms (search strategies) was introduced in Subsection 1.2, in terms of possibility of obtaining the optimum solution and a guarantee of obtaining the optimum in an infinite time.

Here the asymptotic completeness of the process strictly depends on completeness the basic logic synthesis algorithms used, i.e., the synthesis scripts used. However, permutation of variables definitely increases the size of the explored state space and in iterative optimization it helps to avoid local optima. Solutions that were not reached under a given ordering of variable can be reached when permutation is used. The *cordic* circuit is a striking example – it was observed that its optimum solution of 10 LUTs cannot be obtained by ABC from the non-reordered original BLIF (by the synthesis script used here, in any number of iterations). However, a mere permutation allows reaching this optimum.

Generally, the resynthesis with permutations is at least as complete, as are the processes used in the overall synthesis. If there exists a path (under different variable orderings) from the source description to the optimum one, it is guaranteed to be found in an infinite time.

3.4 Conclusions

Experiments presented in this section have shown that the property of synthesis algorithms documented in Section 2 – dependency on the ordering of variables in the initial description – can be advantageously exploited to increase the iterative power of resynthesis.

A positive average improvement in quality (both in area and delay) was obtained. Since introducing the permutations into the iterative process takes almost no time, we can conclude that employing random permutations definitely pays off – random permutations help avoiding local optima. Cases, where worse results are obtained, are relatively rare.

Permutation also offers a possibility of obtaining many *different* solutions, possibly having the same quality (under any quality measure). This feature can be exploited in subsequent synthesis, e.g., a secondary quality criterion may be applied.

More details can be found in [58] and [59].

4 RESYNTHESIS BY PARTS

A resynthesis method will be proposed in this section, where the circuit is iteratively resynthesized by parts only, instead of resynthesizing the circuit as whole – the *resynthesis by parts*. Such an approach may look weird and condemned to be less efficient than resynthesis of the whole circuit, since global information is lost. Surprisingly, this is not the case; it is possible to obtain remarkable improvements (more than 7-times smaller circuits), compared to the standard resynthesis.

The reason for the success is, again, an increase of iterative power and introduction of more of *diversity*. New structures can be discovered by intentionally obscuring the structure of the whole network.

Moreover, even a speed-up of the process may be achieved, because of

1) resynthesizing smaller parts of the circuit is faster than resynthesizing the whole circuit,
2) the process converges faster, thus results of a given quality can be achieved in a shorter time, compared to the classical iterative synthesis.

Note that individual synthesis and optimization algorithms ([15], [24], [25], [28], etc.) process the network by parts as well, because of feasibility limits. However, these parts are relatively small, usually of a "constant" size (4-feasible cuts [24], [25], BDDs with a limited size [15], etc.). In the method proposed in this section, large network parts are resynthesized (up to 90-95%). For smaller parts the method is usually not so efficient.

4.1 MOTIVATION

There has been accidentally encountered an unexpected behavior of logic synthesis in ABC: the LGSynth circuit *e64* [66] was divided into two halves (connected parts) and these were synthesized separately by ABC (by the "`choice`" script [19] followed by standard cells mapping – "`map`"). Then these two parts were merged together. The resulting circuit consisted of 522 gates.

When *the whole* circuit was synthesized using the same process, the resulting circuit had 530 gates. Moreover, the total run-time of this resynthesis was 2.33 seconds, while the total time of the resynthesis of the circuit halves (including the time needed for the circuit splitting) was 1.73 seconds.

This indicates that apparently "something is rotten in synthesis". There *must* exist a case (a sequence of cut/window selections in processes of ABC [24]), where the resynthesis of the whole circuit would be conducted in the same way, as for the separated parts. Moreover, global information is lost in the latter case, thus it theoretically should produce worse results in general.

However, the observed synthesis behavior is not an unusual case. There can be two possible explanations of this unlucky phenomenon:

- low scalability of ABC processes. This means that bigger designs are processed less efficiently;
- heuristics in ABC were accidentally well guided by dividing the circuit.

Hence, possibilities of resynthesizing circuits by parts were investigated more thoroughly. The research resulted in a novel iterative resynthesis method, which will be presented in this section.

4.2 PRELIMINARIES

A Boolean network *N* (circuit) is a structure of connected single-output *nodes* forming an acyclic graph. The network connections, which are naturally inputs and outputs of gates, will be denoted as *signals*.

The network *primary inputs* (PIs) are signals that are driven by the environment; there is no node driving these signals in the network. The *primary outputs* (POs) are signals that drive the environment. Primary outputs may be driving network nodes as well.

The *size* of the network, $|N|$, is the number of its nodes. Primary inputs and outputs are not considered as nodes.

The *fan-in* of a node is the number of its inputs. Since each node input must be driven by exactly one node output in the network (or a PI), the fan-in term will also be used for enumerating the gates driving the respective node. The *fan-out* of a node is the set of nodes it drives. The *transitive fan-in* of a node is a set of nodes that influence the node, i.e., a set of nodes on a path from the PIs to the node. The *transitive fan-out* is a set of nodes that are influenced by the node, i.e., a set of nodes on a path from the node to POs.

The *distance* of two network nodes is the number of signals one needs to pass to reach the other one. The *level* of a node is its maximum distance from any of the primary inputs. Primary inputs have the level equal to 0, by definition.

A *window* is a connected subcircuit N_w of a circuit (Boolean network) N. Formally, it is a Boolean network N_w, $N_w \subseteq N$, whereas for every node $n_i \in N_w$ there exists a path to every node $n_j \in N_w$, $i \neq j$. In the following text, terms *window*, *part*, and *subcircuit* will be used interchangeably, since they have the same meaning in the formal sense.

The aim of the resynthesis is to optimize the quality of the result, be it the network size (e.g. $|N|$), delay (e.g. the number of levels), etc. Let *cost*(N) of the network be the chosen quality criterion, for purposes of this section.

4.3 CIRCUIT RESYNTHESIS BY PARTS

Let us assume an iterative resynthesis process, i.e., a process which can improve the solution when it is run several times consecutively. Let a network N^1 be obtained by running a resynthesis process P on N^0, i.e., $N^1 = P(N^0)$. Subsequent iterations of this process produce different networks, $N^i = P(N^{i-1})$. In an ideal case, $cost(N^i) \leq cost(N^{i-1})$ for every i. However, this may be not true in practice, depending on the executed process.

The proposed *iterative resynthesis by parts* is based on dividing the processed network into two disjoint parts in each iteration, $N^i = N^i_A \cup N^i_B$, $N^i_A \cap N^i_B = \varnothing$, nothing is said about $|N^i_A|$ and $|N^i_B|$ for now. Then one part (N^i_A) is resynthesized, to obtain a functionally equivalent network N^i_R. This network is then merged with the second part (N^i_B), to obtain a new network $N^{i+1} = N^i_R \cup N^i_B$. Obviously, networks N^i and N^{i+1} are functionally equivalent.

4.3.1 THE SYNTHESIS PROCESS

The basic and general principles of the proposed resynthesis process can be described as follows:

```
Resynth_by_parts(Network N) {
    do {
          (N_A, N_B) = Extract_window(N);      // extract the circuit part
          N_R = resynthesize(N_A);             // run the resynthesis
          N' = N_R ∪ N_B;                      // put the network back
          if (cost(N') ≤ cost(N)) N = N';      // any improvement?
    } while (!end());
}
```

Figure 12. The resynthesis by parts algorithm

At the beginning of every iteration, a part N_A of the network (window) is selected and extracted from the original network N. N_B then consists of the remainder of the original network; nodes included in N_A are not present in N_B. Primary inputs and outputs of N (and N_B) are retained, primary inputs and outputs of N_A are determined by the following rules (see Figure 13):

(1) Gate inputs that are not driven by any gate in N_A are assigned as N_A primary inputs (PI_1-PI_5 in the figure).
(2) Gate outputs that do not drive any gate in N_A are assigned as N_A primary outputs (PO_1, PO_2).
(3) Gate outputs that drive some gate in N_B are assigned as N_A primary outputs (PO_3).
(4) Gate outputs that are primary outputs of N are assigned as primary outputs of N_A (PO_4).

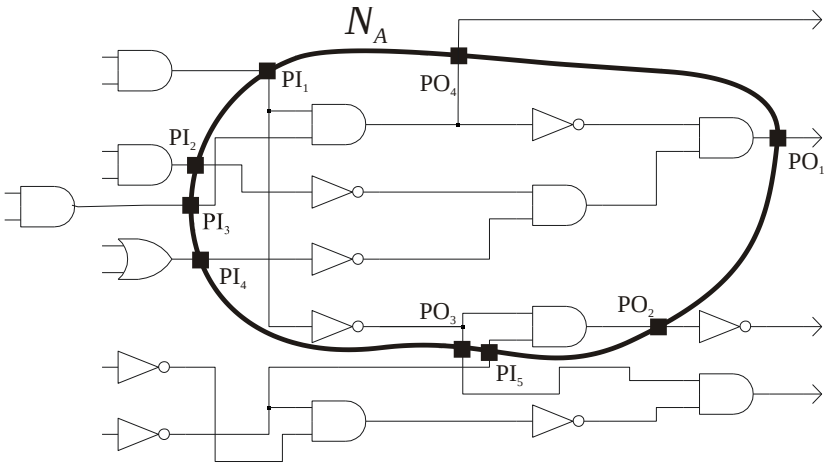


Figure 13. Window Selection

The extracted window N_A is then submitted to ABC synthesis. Any synthesis process may be used in general. In experiments presented here, one iteration of the "`choice`" script is used [19].

The resynthesized network N_R is then merged with N_B. If the resynthesis has brought any improvement, i.e., if the network cost is reduced with respect to the cost of the original network, the old network is discarded and the new one is considered for the next iteration. Thus, the resynthesis is greedy in the "first improvement" sense [36], [38]; solution non-improving iterations are discarded.

Actually, there is a possibility of accepting *any* solution that ABC returns. Under the assumption that ABC will never deteriorate the network, these two alternatives become equal. However, usually this is not the case. Even though such a process usually converges to a stable solution as well, its convergence is naturally slower and produces worse results. Results of these experiments won't be presented here, for brevity.

The whole procedure is iterated, until the stopping condition is satisfied. In experiments presented here, a fixed number of iterations are used, for purposes of comparison. However, more sophisticated stopping criteria should be applied in practice.

4.3.2 WINDOW EXTRACTION METHODS

The **Extract_window** procedure is the essential step in the proposed resynthesis process. Two window extraction algorithms will be described into detail here.

The algorithms are parameterized by the size of the window to be extracted. This parameter value crucially influences the performance of the whole resynthesis run. Details will be discussed in Subsection 4.4.1.

Random Extraction

The **Random Extraction** algorithm is the most naive one. Nevertheless, it gives surprisingly good results. The window (N_A) is gradually constructed by just *randomly* adding nodes, while keeping the window network connected. The algorithm is parameterized by the number of gates of the extracted network (*size*).

The pseudo-code of the algorithm is shown in Figure 14.

```
Random_extract(Network N, size) {
    n = random_node(N);        // select random initial node (pivot)
    N_A = {n};
    N_B = N − {n};             // move it to the extracted network
    while (|N_A| < size) {     // until the required size is reached
        n = random_neighbor(N_A, N_B);    // select random neighbor of any
                                          // node in N_A from N_B
          N_A = N_A ∪ {n};
          N_B = N_B − {n};
    }
    return (N_A, N_B);
}
```

Figure 14. Random window extraction algorithm

Such the most naive approach can be easily modified to minimize the number of the extracted window PIs, POs, or both. However, no significant result quality improvement was observed [72], while the algorithm run-time was significantly increased. Therefore, these techniques were considered inefficient.

RadiusExtract

This algorithm intentionally looks for the *most connected subcircuit*. First, a *pivot* node is selected randomly in the network. Then nodes reachable in a given distance (radius) from the pivot are moved to N_A. In particular, transitive fan-in and fan-out nodes of the pivot are selected, up to a given distance. The algorithm may also be parameterized by the maximum window size, as in the previous case. Thus, the algorithm can operate in two modes, or their combination:

1. If the *size* parameter is set to infinity, all nodes in the given distance (radius) from the pivot node are extracted. The window size is unpredictable and heavily depends on the circuit interconnection density.
2. If the *radius* parameter is set to infinity and the *size* specified, given number of nodes are extracted, whereas the minimum/maximum radius is not guaranteed.

The pseudo-code of the algorithm is shown in Figure 15.

```
Radius_extract(Network N, radius, size) {
    n = random_node(N);                  // start with a random pivot
    q.enqueue(n);                        // use a queue, to ensure
                                         // the FIFO behavior
    while (!q.empty() && |N_A| < size) {
          n = q.pop();
          N_A = N_A ∪ {n};
          for_each_neighbor(m ∈ N_B, n) { // enqueue all neighbors of n
             if (distance(n, m) ≤ radius) // not exceeding the radius
                 q.push(m);
          }
          N_B = N_B − {n};
    }
    return (N_A, N_B);
}
```

Figure 15. The RadiusExtract algorithm

4.4 WINDOW SIZE ANALYSIS AND EXPERIMENTAL RESULTS

The ABC "`choice`" script followed by the "`map`" command, mapping the circuit into library gates was used for resynthesis in the following experiments. A library of all 2-input gates was chosen for simplicity of comparison and low granularity. The "`map`" command is followed by "`sweep`", converting the network of gates into an AIG, so the form of the circuit description is retained. The original benchmark circuits [65] were also mapped into 2-input gates (without any optimization attempts), to generate starting points for the resynthesis.

Let us note here that any synthesis process may be used, without any loss of generality. Any structure-non-destroying resynthesis procedure may be applied, as well as any technology mapping process (standard cells, LUTs, etc.) [72].

If not said otherwise, all the resynthesis processes were run for 5,000 iterations in the experiments. This value is a little bit overkill, since only 2 of the examined 228 circuits needed more iterations to converge [72] using the standard synthesis process. However, it enables us to compare rather stable solutions and measure the convergence of the processes more precisely.

4.4.1 INFLUENCE OF THE WINDOW SIZE

The influence of the window size on the result quality (in terms of the number of 2-input gates) will be investigated in this subsection.

First, the influence of the window size, relative to the resynthesized circuit size will be examined, for the Random Extraction algorithm. The window size was varied from 10% to 100%, for all the 228 circuits. Average improvements obtained w.r.t. the state-of-the-art, i.e., the repeated resynthesis of the whole circuit, were computed. The results are shown in Figure 16.

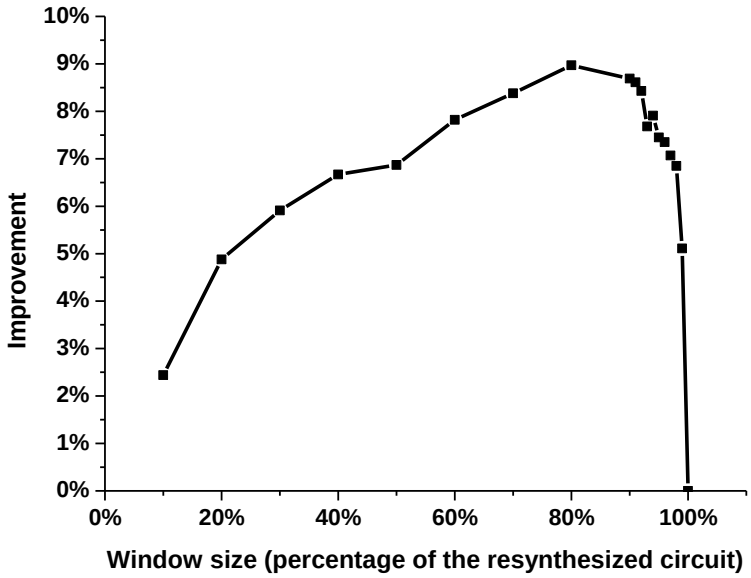


Figure 16. Influence of the window size – Random Extraction

We can see that the maximum improvement is achieved for window sizes ranging from 80 to 90% of the resynthesized circuit. If bigger windows are extracted, the quality of the result quickly drops.

Notice that the 100% limit case exactly corresponds to the approach presented in Section 3. Even though the extracted network is equal to the original one, PIs and POs are randomly reordered, from the gradual nature of the algorithm (nodes are selected randomly, and thus also PIs and POs are randomly reordered, according the instant they appear in the window). The only difference is that in Section 3 even deteriorating solutions are accepted and the best solution ever obtained is returned as the final result. In resynthesis by parts only non-deteriorating solutions are accepted.

The average improvement, as a function of the relative window size for the RadiusExtract, is shown in Figure 17.

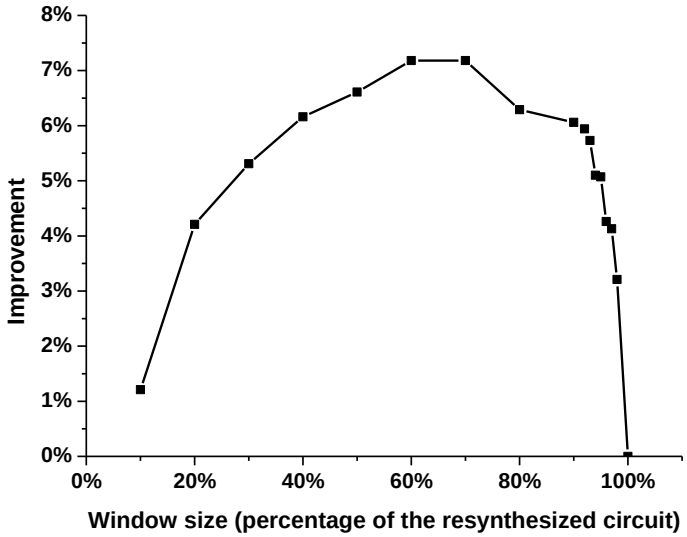


Figure 17. Influence of the window size – RadiusExtract

A similar behavior can be observed here: large windows produce inferior results. However, the maximum improvement is achieved for much smaller windows, compared to the Random Extraction method. This is because of the RadiusExtract method produces more compact windows. Conversely, in Random Extraction many “useless” gates are included in the window, thus the window naturally must be larger to obtain comparable results.

The dependency of the improvement ratio on the window radii is shown in Figure 18. Here we see clearly, that best results are obtained for radii ranging from 5 to 7. Higher radius values produce inferior results, in most cases because of the window starts spanning the whole circuit, i.e., results of 100% resynthesis are obtained.

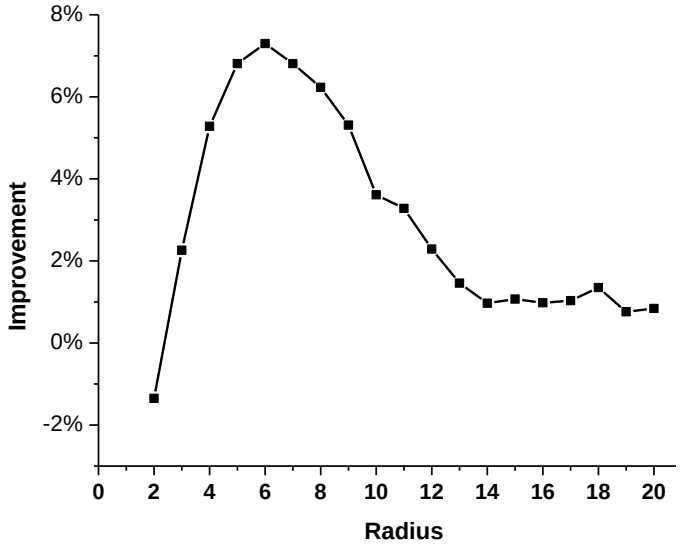


Figure 18. Influence of the window size – RadiusExtract

Now, when there are two metrics of window sizes and a clear dependency of the result quality on the window size is seen, we may ask if the metrics are related somehow.

An experiment was performed to answer this question: the percentage window sizes generated by the RadiusExtract method were observed. The histogram of frequencies of window sizes, for the *misex3* benchmark circuit [65] is shown in Figure 19. The radius was chosen to be 6, as the most promising one (see Figure 18).

We can see clearly that the maximum of window sizes lie between 60-90%. Similar behavior can be observed for most of circuits. This fully conforms to the observation in Figure 17. Therefore, we can conclude that the algorithm behaves *consistently*, even when the influence of random factor is high.

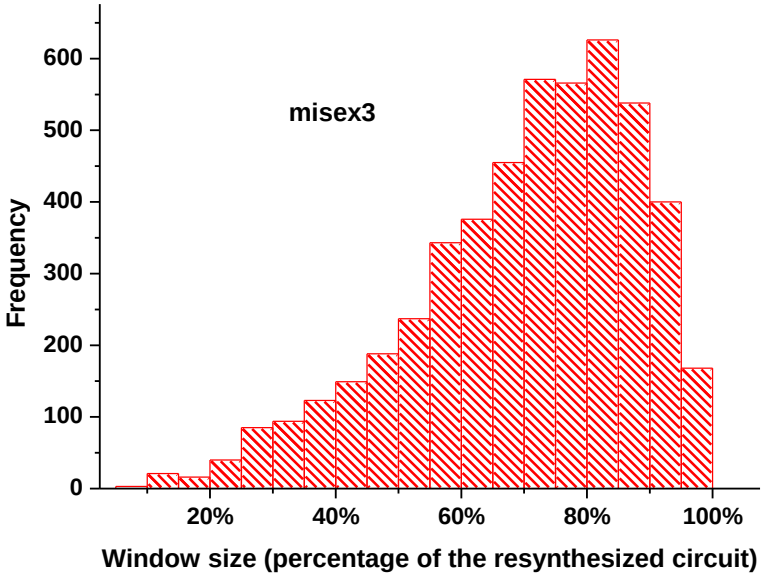


Figure 19. Window sizes generated by RadiusExtract

It can also be observed that the extracted window sizes *scale* with the resynthesized circuit size, when having a constant radius. This is a rather surprising observation, since the window extraction procedure is strictly local. This could be probably explained by the fact that the extracted window often prematurely reaches "the borders" (i.e., PIs, POs) of the circuit for smaller circuits. The scatter graph illustrating the dependency is shown in Figure 20 for all the 228 circuits resynthesized for 5,000 iterations, radius 6.

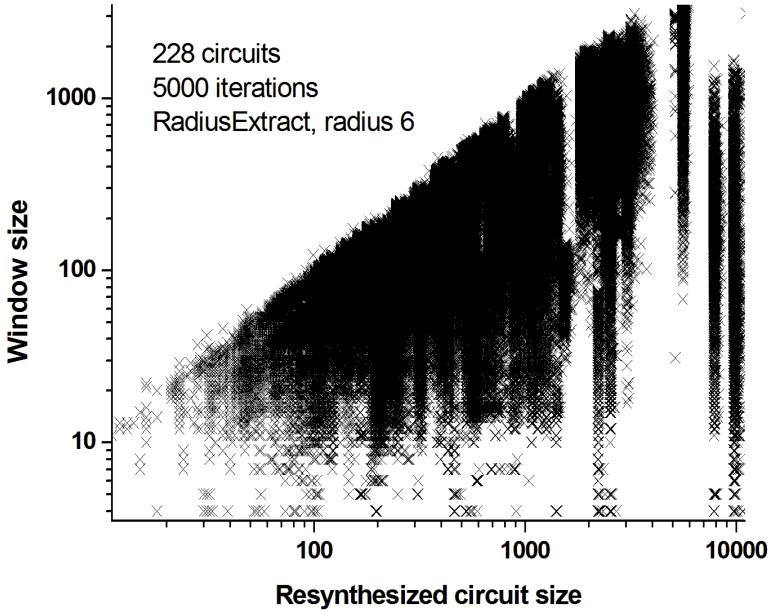


Figure 20. Extracted window sizes

4.4.2 COMPARISON WITH STANDARD SYNTHESIS

A comparison with the state-of-the-art, i.e., the iterative resynthesis of the whole and non-permuted circuit will be shown here. Results of 10 largest circuits from the 228 ones [65] are shown in Table 7. All the iterative processes were run for 5,000 iterations.

After the benchmark name, its original size in terms of 2-input gates is given (*"orig."*). Then the number of gates obtained by 100% resynthesis is shown ("*100%*"). The "*conv. iters.*" column gives the number of iterations ABC needed to reach the final solution, thus possibly converge to a stable solution. However, very high values indicate that probably even better solutions could be reached, if iterated further (more than 5,000 iterations).

Numbers of gates obtained by the RadiusExtract and Random Extraction methods and percentage improvements w.r.t. the 100% resynthesis follow.

The "*eq. iters.*" columns indicate the numbers iterations needed to reach a solution of at least the same size as the one obtained by 100% resynthesis.

Radius 6 and 80% circuit parts were chosen, for the RadiusExtract and Random Extraction, respectively. The summary (for numbers of gates) and average (for percentages and #of iterations) results are presented in the last table row.

Table 7. Benchmarks results

name	*orig.*	*100%*		*Radius 6*			*Random 80%*		
		gates	*conv. iters*	*gates*	*impr.*	*eq. iters*	*gates*	*impr.*	*eq. iters*
s38584.1	11,210	9,752	1,342	9,692	0.6%	2,187	9,735	0.2%	1,138
s38417	8,643	7,891	1,934	7,834	0.7%	808	7,883	0.1%	261
prom1	6,220	5,829	3,769	5,548	4.8%	11	5,562	4.6%	48
too_large	4,182	3,033	2,467	3,129	-3.1%	N/A	2,746	9.5%	215
misex3	3,539	2,645	4,147	2,362	10.7%	2,909	1,970	25.5%	179
mainpla	3,472	3,091	4,215	3,027	2.1%	481	2,958	4.3%	19
apex2	3,394	2,083	41	1,998	4.1%	3,165	1,786	14.3%	275
des	3,158	2,915	1,233	2,815	3.4%	74	2,746	5.8%	39
xparc	2,930	2,540	396	2,406	5.3%	108	2,363	7.0%	14
seq	2,771	2,024	2,161	1,803	10.9%	1,157	1,707	15.7%	129
Sum/avg.	**136,755**	**117,215**	**398.2**	**110,923**	**7.3%**	**102.0**	**109,335**	**9.0%**	**49.5**

We can see that resynthesis by parts, both RadiusExtract and Random Extraction, almost always produces better results than 100% resynthesis. Moreover, also a possible speedup can be seen – resynthesis by parts reaches the same solution as 100% resynthesis in significantly less time (8-times, on average, for the Random Extraction case).

The results obtained from all the 228 examined benchmarks are shown in Figure 21. The scatter-graph visualizes the improvement achieved by the resynthesis by parts (Random Extraction, 80%, 5,000 iterations), as a function of the original circuit size (in terms of 2-input gates). Notice the logarithmic x-axis. The highest improvements are achieved for mid-size circuits here, however significant improvements can be seen even for larger circuits. Improvement was achieved for a vast majority of circuits, the occasional deterioration was not higher than 7%.

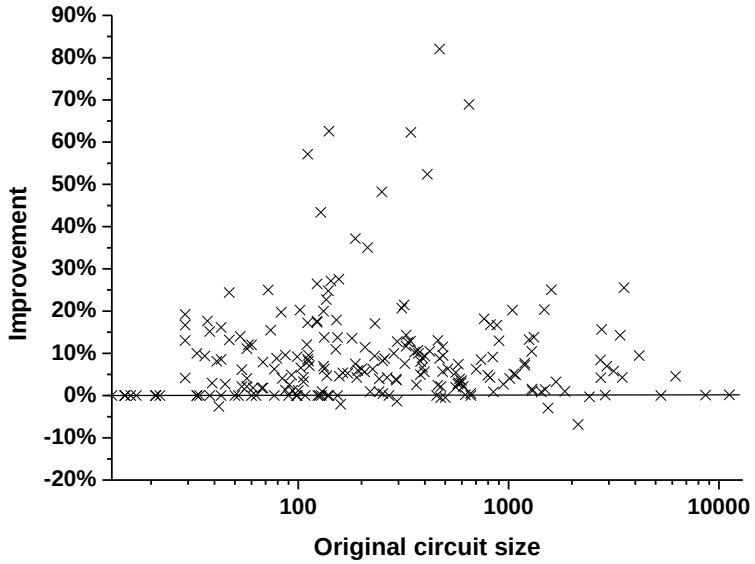


Figure 21. Summary results

4.4.3 ITERATIVE POWER

As in the method presented in Section 3, the main cause of the observed success is an increase of the iterative power, w.r.t. the 100% resynthesis. Thus, the iterative power of the resynthesis by parts will be investigated into detail.

The example analysis will be performed on the *misex3* circuit [65]. This circuit belongs to the "hardest" ones, since even the 100% resynthesis converges rather slowly. In particular, it needs more than 4,000 iterations to converge to a stable solution.

The Random Extraction was used, window size 80%. The resynthesis was run 20-times and the progress of the solution and the span in the result quality was observed.

The convergence curves are shown in Figure 22. The topmost curve belongs to the 100% resynthesis case, the curves obtained from 20 re-runs of resynthesis by parts are seen below. We can see that the 100% resynthesis has never outperformed the resynthesis by parts. Even though the resynthesis process also tends to get stuck in a local minimum, it converges longer, which enables the synthesis reach much better results.

Such a behavior can be observed for all circuits "difficult" for synthesis. For "easy" [49], [50] circuits the global optimum is found quickly by both methods indifferently. These circuits can be seen in Figure 21, on the 0% improvement line.

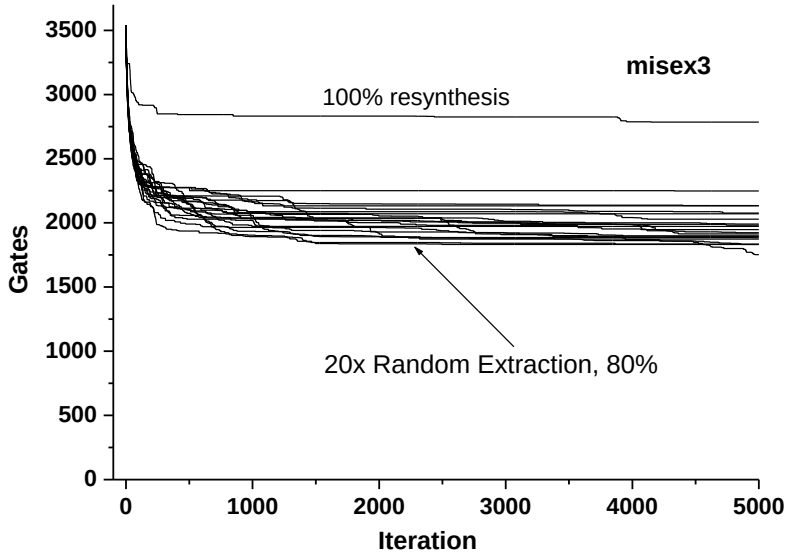


Figure 22. Convergence curves for *misex3*

4.5 RANDOM NUMBER GENERATOR GRANULARITY EFFECTS

Since the resynthesis by parts is a randomized process, we may ask how much randomness is indeed needed, to make it perform sufficiently [58]. Particularly, it is worth studying of how much is the process influenced by the random number generator *granularity*, i.e., the number of different values it produces. Note that the random number generator granularity studied here has nothing in common with the random number generator *quality*, i.e., its period.

As a result, we will obtain some theoretical conclusions: we ask what will happen if a randomized algorithm will be rendered completely deterministic and if the algorithm *could* be deterministic, actually.

For the purpose of this study, the algorithm was *partially derandomized*. Partial derandomization was made by modifying the random number generation function, so that it produces only 1, 2, 3, etc. distinct values, while still keeping its period. In the 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, a constant 0. For *RF* = 2, two border values are produced (0 and the maximum), etc. For *RF = infinity* the unmodified random number generator is used.

Note that this kind of derandomization has nothing to do with known derandomization techniques to derive a completely deterministic algorithm from a randomized one [74]. Derandomization used throughout this work will be understood just in sense of reducing the amount randomness, not removing randomness completely, or as an attempt for a design of a deterministic algorithm.

Convergence curves for the "*e64"* [65] circuit, are shown in Figure 23. The Radius Extraction method with radius 6, 5,000 iterations was used. All the data was obtained by averaging results of 20 independent runs, to make the results more precise.

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 approaches the *RF* = *inf.* one, for *RF* = 100 the curves already blend (not shown in the Figure).

Let's note that one may think that in the case of *RF* = 1 only a constant part of the circuit is resynthesized, leaving the rest of the circuit unmodified. However, this is not the case, because of the algorithm implementation. Even though the random number generator always selects the first network node (in terms of the program internal network structure) as the pivot, the extracted and resynthesized network is always appended *to the end* of the structure. Therefore, the next pivot will be the first node of the yet unprocessed network. Consequently, all nodes have a chance for resynthesis. And what's more, all nodes have an *equal* chance for being processed (up to the given number of iterations limit, of course). Therefore, the case of *RF* = 1 becomes a *systematic*, *deterministic*, and *fair* way of resynthesis by parts. The results show that this is the worst possible way.

The necessary random number generator granularity needed can be derived analytically as well. The random choice occurs in the pivot selection procedure only. 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.

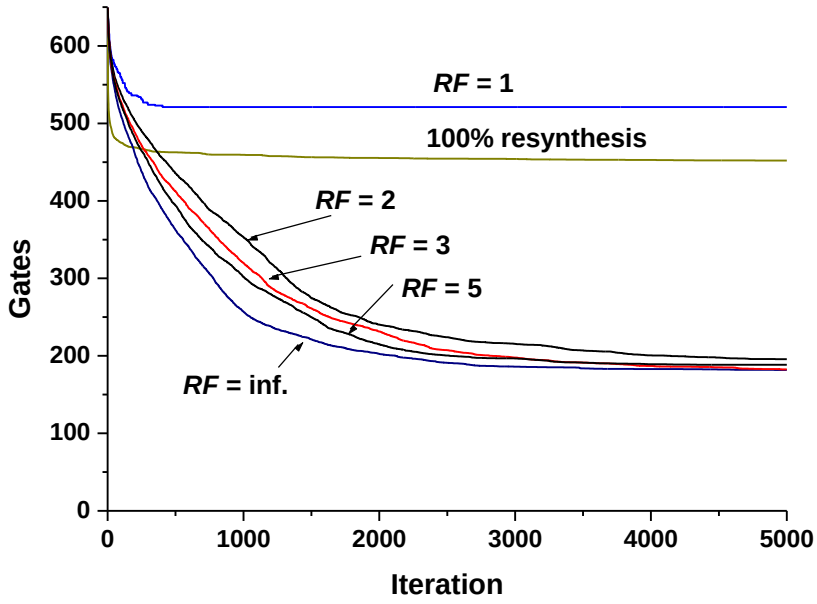


Figure 23. Resynthesis by parts – derandomized (*e64*)

4.6 Asymptotic Completeness of the Algorithm

As in the previous section, completeness of the overall algorithm strictly depends on completeness of the basic synthesis algorithms used. But still, the size of the state space explored is extended even more, compared to using permutations only; structures that could mislead the basic synthesis algorithms can be theoretically avoided, by obscuring their parts.

4.7 Conclusions

The notion of high-level iterative randomized resynthesis based on permutation of variables was further extended to *resynthesis by parts* of the circuit. Even more randomness is introduced this way, which is documented by the presented results. Actually, the technique presented in Section 3 (introducing different variable orderings) can be understood as a special case

of resynthesis by parts, where 100% parts with random orderings of variables are resynthesized. Significant benefits over this simple method were shown here.

As well as in the permutations-based approach, randomized nature of the algorithm allows generation of different solutions, which can be exploited by further synthesis. Moreover, the used resynthesis process can be "externally" influenced by cutting the circuit into parts, which, in consequence, can also force the process generate different structures.

Values shown in Table 7 indicate that resynthesis by parts is able to reach results of equal quality as the 100% resynthesis in less iterations (compare the "*conv. iters.*" and "*eq. iters.*" columns).

Concluded, the proposed method is able to produce *better* results than the iterative state-of-the-art and it produces them *faster*.

The results suggest that resynthesis by parts is *always* able to produce results of at least equal size as the deterministic 100% resynthesis, when run long enough – the employed randomness and circuit separation successfully prevents getting stuck in local minima.

All the experiments were performed using a constant limit of the number of iterations, which was for most of circuits apparently unnecessarily high. However, this was the intention – comparative and representative results are obtained this way; the behavior could be studied more thoroughly. In practice, an adaptive stopping mechanism should be applied.

The scalability of the process is unquestioned. The window sizes scale with the design size linearly, thus the expected behavior of the resynthesis is retained even for large designs.

If implemented optimally, the resynthesis by parts introduces only a negligible time overhead, compared to standard synthesis processes. Therefore, this proposed method is expected to perform at least as well as the state-of-the-art synthesis for any design sizes.

Experiments have shown that the resynthesis by parts cannot be successfully performed as a deterministic process. However, only very little of randomness is necessary for a success.

More details can be found in [72], [73], and [58].

5 BOOM – THE SOP MINIMIZER

Another example of a randomized iterative optimization process is a two-level (SOP) minimizer BOOM [75], [76]. Randomness is used when *more equally valued decisions* are available.

Iteration is performed at a high level too, even though a solution needs not necessarily be obtained after every iteration (see Subsection 5.2). The two-level minimization is run repeatedly, whereas a new set of implicants covering the source function is produced in each iteration. Since the implicant generation phase is randomized, there is a big chance of producing *different* implicants in each iteration. The final solution is then constructed by combining all the obtained implicants. A solution may be generated already after the first iteration. However, the more implicants are available, the better solution can be obtained. Therefore, here we see a possibility of trade-off between the run-time and quality as well as in the algorithms described in Section 3 and Section 4.

The algorithm will be briefly described in the following subsections.

5.1 PRELIMINARIES

Let us have a set of m Boolean functions of n input variables $F_1(x_1, x_2, \ldots x_n)$, $F_2(x_1, x_2, \ldots x_n)$, $\ldots$ $F_m(x_1, x_2, \ldots x_n)$. Values of input variables, for which F_i evaluates to 1 will be called the *on-set* $F_i(x_1, x_2, \ldots x_n)$, values, for which F_i evaluates to 0 will be called the *off-set* $R_i(x_1, x_2, \ldots x_n)$, and values, for which the value of F_i may be arbitrary (0, 1) will be called the *don't care set* $D_i(x_1, x_2, \ldots x_n)$. BOOM needs to have the on-sets and off-sets of all m functions explicitly specified. The don't care set is then specified implicitly, as the complement of the union of the on- and off-set.

The on- and off-sets are described by sum-of-products (SOP) forms, or more particularly, by a truth table (in the PLA format [6], [66]).

Enumerating the two care sets instead of the on-set and don't care sets (which is usual, e.g., in the MCNC benchmarks [66] and also in ESPRESSO [6]), is more practical for problems with a large number of input variables, because here the size of the don't care set greatly exceeds the two care sets in practical cases. The need of minimizing such functions ([77], [78], [79], [80], [81], [82]) was the motivation for developing BOOM.

In the SOP form, the product terms will be denoted simply as *terms* or *cubes* in the further text. Terms having n literals will be denoted as *minterms*. The *dimension* of a term is the number of its non-assigned variables (DC variables), thus, it is computed as *n-#*of term literals. Minterms have the dimension equal to zero.

The set of terms covering the on-sets of all functions will be denoted as the *cover*.

The minimization task is to find the minimum cover, i.e., to produce SOP forms $G_i(x_1, x_2, \ldots x_n)$, $1 \le i \le m$, while $F_i \subseteq G_i$ and $G_i \cap R_i = \emptyset$. The optimization (quality) criterion is usually the total number of SOP terms in G_i. Terms can also be shared among different G_i's (group minimization). Secondary quality criteria are usually the number of SOP *literals* (occurrences of input variables in the SOP forms) and the *output cost* (the total number of terms, if no sharing of terms is allowed). In the AND-OR two-level implementation of the SOP, the number of literals reflects the complexity of the AND plane, the output cost reflects the complexity of the OR plane.

5.2 THE BOOM ALGORITHM

A self-explanatory pseudo-code of the overall algorithm is shown in Figure 24. Before executing the algorithm, the on-set of the source function must be provided. In case of one set missing, it is computed as a complement of the two other sets [6].

The *CD-Search* (Coverage-Directed Search) is the vital phase of BOOM (Subsection 5.2.1). It produces the initial cover (set of implicants) of the source on-set (F_i). This is also the phase where randomness is used the most. These implicants are stored in the *implicant pool*. Here all the implicants ever obtained during the minimization process are accumulated. Internal structure of the pool does not allow duplicities, thus only unique terms are stored there [75], [83].

The implicants are further expanded (Subsection 5.2.2) to form prime implicants, while the original (non-prime) implicants are not discarded. Then all the implicants are reduced (Subsection 5.2.3) to obtain group implicants, i.e. implicants of more output functions. Again, no implicants are discarded; only the new ones are added to the pool.

At the end of each iteration the pool is "purged" by resolving clear dominance relations [84]. Thereby, apparently redundant terms are removed.

This process is repeated, until the stopping criterion is met. This is usually a user-specified maximum number of iterations, timeout, or the desired solution quality.

The solution is then formed by solving the covering problem (Subsection 5.2.4) using all the implicants in the pool – an irredundant subset of implicants covering the on-sets of all functions is formed.

Finally, the solution is tried for the final refinement (Subsection 5.2.5) to keep only necessary group implicants, which are then further expanded. This phase is similar to the ESPRESSO's make_sparse procedure [6].

Note that the UCP solution and Sparse can be executed inside the main iteration loop too. This would be necessary, e.g., if the stopping condition should be determined based on the solution quality. Then BOOM can be considered as a fully high-level iterative process. Otherwise, when the stopping criterion is, e.g., a fixed number of iterations, the UCP and Sparse can be run only once, as in Figure 24.

```
BOOM(F, R) {
    Pool = ∅;
    do {
          Cover = CD-Search(F, R);
          Pool = Pool ∪ Cover;
          Pool = Pool ∪ Cover.Expand(R);
          Pool = Pool ∪ Cover.Reduce(F, R);
          Pool.Purge();
    } while (!stop());
    Solution = UCP_Solve(F, Pool);
    Solution.Sparse(F, R);
    return Solution;
}
```

Figure 24. BOOM algorithm – pseudo-code

5.2.1 COVERAGE-DIRECTED SEARCH

The *Coverage-Directed Search* (CD-Search) is the first and the most important step of the BOOM algorithm. Here the initial cover of the on-set is generated. The main idea here is the *top-down* approach to generation of implicants. Classical SOP minimizers [6] start with the original cover, which is further refined. This may cause the algorithm to fail, if the original description is "unsuitable", i.e., if a deep local optimum in the state space is given as the source.

In BOOM, the implicants are not derived from the original cover directly; the original cover is used just as a "clue". The off-set also serves just as a constraint. The algorithm starts with an *n*-dimensional cube (i.e., an all-DC term, a term with zero literals), which is obviously not an implicant of any function, unless the off-set is empty. This term is gradually reduced by adding literals to it, until it becomes an implicant (does not intersect the off-set).

The output functions F_i, $1 \leq i \leq m$, are processed separately, one by one – no group minimization is considered in CD-Search. Therefore, single-output functions will be assumed in this and the following subsection, and thus no ambiguity of terms like on-set, off-set, implicant, can occur.

The search for terms is directed towards finding an implicant that covers as many on-set terms as possible – from this comes the algorithm name. The implicant generation starts with selecting the *most frequent literal* from the given on-set. It forms an (n-1) dimensional (1-literal) cube possibly covering most of the on-set terms.

This cube is then checked against the off-set, by a simple pair-wise comparison of the cubes. If the cube does not intersect any off-set term, it is an implicant. If the off-set is intersected, another most frequent literal is added and the term is checked again.

After each literal selection the terms that cannot be covered by any term containing the selected literal are temporarily removed from the on-set, for a more efficient search. These are the terms containing that literal with the opposite polarity.

When an implicant is finally obtained, it is recorded (stored in the pool) and the on-set terms that are covered by this term are removed from the on-set. Thus, a reduced on-set containing only yet uncovered terms is obtained. Now the whole procedure is repeated from the beginning. The search for implicants continues, until the whole on-set is covered.

The output of this algorithm is a set of product terms covering the whole on-set and not intersecting any off-set term.

The basic CD-Search algorithm for a single-output function can be described by the pseudo-code shown in Figure 25. The on-set (F) and the off-set (R) are the inputs to the algorithm; the output is the sum of products (G) that covers F and does not intersect R.

Note that this algorithm may be further enhanced by more sophisticated literal selection techniques whose description, however, exceed the scope of this text. For details see [75].

```
CD_Search(F, R) {
    G = ∅;                  // G is being created
    do
          F′ = F;           // F′ is the reduced on-set
          t = 1;            // t is the term in progress
          do {
                v = most_frequent_literal(F′);
                t = t.Add(v);
                F′ = F′ – cubes_not_including(t);
          while (t ∩ R ≠ ∅);
          G = G ∪ t;
          F = F - F′;
    until (F == ∅);
    return G;
}
```

Figure 25. CD-Search

The point, where randomness takes place, is the literal selection process. As stated above, the primary criterion is the *maximum occurrence* in the yet uncovered on-set. However, usually many equally valued choices occur in practice (see Subsection 5.7). Then one is chosen randomly.

Note that one particular literal selection has a crucial impact on the execution flow of the rest of the algorithm, since literal-specific restrictions of the on-set are applied. Therefore, there is a big chance that many different solutions will be obtained, depending on random decisions.

To introduce even more randomness to CD-search, *mutations* may be present. With a given probability, a mutation occurs. Then a literal with any non-zero frequency of occurrence is selected, instead of the literal with the maximum one. We have found experimentally, that 2-5% of mutations are beneficial. For details see [83].

5.2.2 IMPLICANT EXPANSION

Implicants obtained by CD-Search need not be prime. Hence, they should be further expanded into primes, by removing literals and checking against the off-set. Basically, each term is processed separately and literals are tried for removal one by one. After each literal removal, the term is checked if it is still an implicant, again by comparing it with the off-set. When no literal can be removed without making the term intersect the off-set, it is a prime implicant.

There may be several strategies of literal removal. Essentially, there are $O(2^n)$ possibilities of expansion; $O(2^n)$ different prime implicants can be obtained from one term. Performing such an exhaustive implicant expansion is usually infeasible in practice.

Therefore, a simple greedy (and again randomized) strategy is used: all the literals are tried for removal *sequentially*, i.e., one by one, starting from a random position. One prime implicant is obtained from one non-prime this way and the algorithm time complexity is linear with n. This strategy may be further extended to multiple random restarts, where $O(n)$ primes is produced, for a $O(n^2)$ complexity expense.

5.2.3 IMPLICANT REDUCTION

All the obtained prime implicants are tried for reduction by adding literals to them, in order to become implicants of more than one function (F_i, $1 \leq i \leq m$). The principle of the method of implicant reduction is similar to the implicant expansion one (Subsection 5.2.2) and, indeed, the CD-Search as well (Subsection 5.2.1). Literals are gradually *added* to the previously obtained implicants, until there is no chance that the term will become an implicant of more functions. This is checked by comparing the reduced term with off-sets of all functions (R_i, $1 \leq i \leq m$); if a term does not intersect the off-set of F_i, it is an implicant of F_i.

Preferably, literals that prevent intersecting with most of the terms of the off-sets of all F_i (i.e., yielding reduced terms covering the least zeros in all the m functions) are selected.

All implicants that were ever found in this phase are stored in the pool and outputs are assigned to them – it is checked for each term produced, what functions from F_i, $1 \leq i \leq m$ it is an implicant of.

Randomness is applied here as well, in the CD-Search manner. When there are two or more equally valued choices, one is taken randomly.

5.2.4 COVERING PROBLEM SOLUTION

All the different implicants are finally entering the unate covering problem (UCP) [37], [84] solution process. Here their subset is selected, to form an *irredundant cover* of the on-set (F_i, $1 \leq i \leq m$). The result of this phase may serve as a solution of the minimization problem.

Even though very efficient exact UCP solvers exist ([85], [86]), there is still a danger of exponential run-time explosion, since the UCP problem is NP-hard [37]. This is emphasized even more by the solved problem sizes; typically thousands of implicants are entering UCP. Therefore, a simple greedy heuristic algorithm is used in practice [75].

First of all, simple dominance relations are resolved [84], to significantly prune the pool. Then, a greedy heuristic is applied. The solution is constructed by gradually adding implicants to it, until the whole on-set is covered. The heuristic has several decision stages, where the candidate implicants are gradually filtered out:

1. Select implicants covering most of yet uncovered on-set terms.
2. From these, select implicants covering on-set terms that are difficult to be covered (they are covered by the minimum of implicants).
3. From these, select the ones with the least cost (the number of literals).
4. If there are still more possibilities, choose one *randomly*.

Here we see the randomness as well – when there are more than one decision at the last stage of the filtering process, one is taken randomly.

5.2.5 THE FINAL SIMPLIFICATION

The solution obtained by UCP can be further slightly simplified. The number of terms must definitely stay unmodified; however, the number of literals and the output cost (see Subsection 5.1) can still be reduced for multi-output functions with shared (group) implicants.

Before the UCP phase is entered, each term is assigned a set of *all* functions (from F_i, $1 \leq i \leq m$) which it is an implicant of. A particular (group) implicant can be necessary for obtaining a solution of one function, however it needs not be (but can be) present in a solution of another functions it is an implicant of too. Its presence in the final solution then just increases the output cost.

Therefore, the UCP is solved upon the obtained cover for each of the *m* functions *separately*, to find the set of really necessary implicants for each. As a result, the output cost is reduced.

Moreover, when the "demands" on implicants were loosened now, it may happen that it will be possible to further expand them. Hence, the implicants in the solution are further tried for expansion (see Subsection 5.2.2).

The result obtained by this phase is a *prime* and *irredundant* cover [6] of the on-sets of all the source functions. Moreover, it is prime and irredundant also in terms of the number of literals and the output cost. Of course, the minimality is not guaranteed, since all the employed algorithms were greedy.

5.3 THE ITERATIVE MINIMIZATION

As stated above, BOOM can be (but not necessarily) run in an iterative way, in order to improve the result quality. The more implicants are accumulated, the better final solution can be expected.

The progress of iterating the process is illustrated in Figure 26. A randomly generated function of 20 input variables, 5 output variables, 200 care terms of average dimension 2 was minimized here. A random function was chosen for this experiment, in order to maximally suppress the influence of any possible singular behaviors of standard benchmark circuits ([66]).

The graph shows the progress of minimization in course of iteration. The total number of implicants in the pool is depicted by the thin line (and the left x-axis) and the solution quality (UCP was solved after each iteration, for the example purposes), in terms of the total SOP 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 [6] is indicated as a horizontal hairline. It can be seen that even though rather inferior solutions are produced in early iterations, BOOM overcomes ESPRESSO in the solution quality in the 144-th iteration. This result can be generalized for almost any non-trivial circuit. In cases where ESPRESSO does not produce optimum results, BOOM is able to obtain them for a possible expense of run-time (see Subsection 5.4).

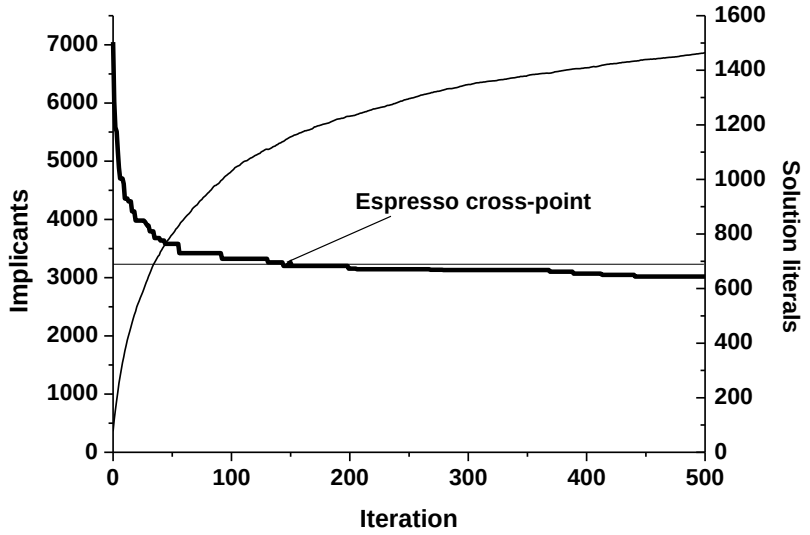


Figure 26. Iterative process in BOOM

5.4 COMPARISON WITH ESPRESSO

The performance of BOOM will be evaluated here, in comparison with ESPRESSO. Both random and practical benchmarks will be tested.

All the BOOM experiments were run on a standard PC with a 900 MHz Athlon CPU and 256 MB of RAM.

5.4.1 MCNC BENCHMARKS

The BOOM algorithm was tested on standard MCNC benchmarks [66] and the results and run-times were compared to ESPRESSO [6]. BOOM was run for one iteration only in this experiment, to illustrate its basic capabilities. Indeed, no more iteration was necessary to reach competitive results.

The results of selected benchmarks are shown in Table 8. The benchmarks were also solved by ESPRESSO-EXACT in order to obtain the minimum solution for comparison. Note that the minimality criterion for ESPRESSO-EXACT is the number of terms only, and thus some "exact" solutions are even worse in the literals counts than those reached by ESPRESSO or BOOM. Some benchmarks were not solved by ESPRESSO-EXACT because of its extremely long run-times (blank entries in Table 8). ESPRESSO solutions that are equal to the exact ones are shaded in the ESPRESSO column. The column *n/m/p* describes the numbers of input/output variables and care terms of the benchmarks, the *time* columns indicate the computational time in seconds, the *lit/out/terms* columns show the quality of the results, i.e., the number of literals in the final SOP form, the output cost and the number of terms. The shadowed cells indicate that the benchmark was solved by BOOM in a shorter time than by ESPRESSO, or the same result was reached respectively.

As the MCNC benchmark circuits mostly have a relatively low number of inputs and many care terms defined, the iterative features of BOOM couldn't be fully exploited here. Thus, the results are not optimal comparing to ESPRESSO. However, BOOM is much more efficient for more complex problems (see the following Subsections), which ESPRESSO often cannot solve in a reasonable time.

Table 8. Runtimes and minimum solutions for the standard MCNC benchmarks

		ESPRESSO		ESPRESSO-EXACT		BOOM – 1it.	
bench	*n/m/p*	*time*	*lit/out/terms*	*time*	*lit/out/terms*	*time*	*lit/out/terms*
alu2	10/8/241	0.07	268/79/68	0.18	268/79/68	0.02	268/79/68
alu3	10/8/273	0.08	279/70/65	0.19	278/74/64	0.02	279/68/66
alu4	14/8/1184	0.59	4445/644/575	12.24	4495/648/575	1.02	4449/636/577
b9	16/5/292	0.08	754/119/119	0.89	754/119/119	0.09	754/119/119
br1	12/8/107	0.05	206/48/19	0.07	206/48/19	0.02	215/45/20
br2	12/8/83	0.06	134/38/13	0.07	134/38/13	0.01	134/38/13
chkn	29/7/370	0.14	1598/141/140	0.25	1602/142/140	0.41	1598/141/140
cordic	23/2/2105	1.86	13825/914/914	3.59	13843/914/914	4.05	13825/914/914
ex4	128/28/654	0.62	1649/279/279			14.01	1649/279/279
e64	65/65/327	0.11	2145/65/65	0.11	2145/65/65	15.06	2145/65/65
exep	30/63/643	0.17	1175/110/110	0.55	1170/108/108	3.66	1175/110/110
ibm	48/17/499	0.11	882/173/173			0.82	882/173/173
mark1	20/31/72	0.25	97/57/19	1.45	97/57/19	0.04	93/46/23
misex2	25/18/101	0.07	183/30/28	0.06	183/30/28	0.10	183/30/28
misex3c	14/14/1566	0.98	1306/253/197			0.59	1335/242/209
misj	35/14/55	0.07	54/48/35			0.03	54/48/35
shift	19/16/200	0.07	388/105/100			0.06	388/105/100
spla	16/46/837	0.71	2558/643/251	6.65	1564/450/181	1.54	2821/517/285
vg2	25/8/304	0.08	804/110/110	0.54	804/110/110	0.15	804/110/110
x9dn	27/7/315	0.08	1138/120/120	0.49	1138/120/120	0.22	1138/120/120

5.4.2 RANDOMLY GENERATED BENCHMARKS

Randomly generated benchmarks (PLAs) were chosen for the following experiments for several reasons:

- BOOM is efficient especially for functions with a large number of inputs and few care terms. Such are available in standard benchmark suites [66] only rarely;
- PLAs coming from some practical problems, which BOOM was intended for [80], [81], mostly are of a random nature;
- randomly generated benchmarks allow for scaling, thus also the scalability of BOOM can be determined.

A set of problems having up to 200 input variables and up to 200 care terms was solved. The truth tables (PLAs) of these problems were generated randomly, while only the number of input variables and the number of care terms were specified. The number of outputs was set equal to 5, and the terms contained 20% of don't cares (missing input variables in the terms). The on-sets and off-sets of each function were kept approximately of the same size. For each problem size (number of variables, number of terms), ten different samples were generated and solved and average values of the ten solutions were computed.

The results are shown in Table 9, Table 11, and Table 11. Here the number of input variables *n* increases horizontally and the number of specified PLA terms *p* is increased vertically. The first row of each cell contains BOOM results, the second row shows ESPRESSO results. The quality criterion selected for BOOM was the sum of the number of literals and the output cost, which well approximates the final implementation cost using standard library gates.

First, the minimality of the result was compared. BOOM was run iteratively, using *the same run-time* as ESPRESSO needed to obtain the solution. The results are shown in Table 9 and Table

11. These two tables contain results of the same experiment, only results details are given in Table 11, whereas Table 9 contains the optimization criterion values (sum of the number of literals and the output cost).

The numbers of iterations BOOM was run is shown in the parentheses in the first line entries, while the run-time is indicated in the ESPRESSO entries (second lines).

We can see that for all but one problem size (shaded cell) BOOM found a better solution than ESPRESSO in the same time.

Table 9. Randomly generated PLAs – comparison of the result quality (same time as ESPRESSO)

p / n	50	100	150	200
50	151 (58) 176/3.89	131 (90) 149/10.29	122 (147) 133/24.87	113 (199) 128/41.99
100	370 (46) 393/9.31	297 (94) 315/77.07	278 (140) 293/199.17	264 (140) 275/246.21
150	606 (43) 639/54.76	490 (101) 509/282.80	454 (116) 458/646.20	471 (64) 429/1066.14
200	855 (51) 895/162.62	690 (116) 704/730.91	617 (207) 629/1913.65	579 (277) 586/3372.66

Entry format: BOOM: # of literals + output cost (# of iterations)
ESPRESSO: # of literals + output cost / time in seconds

Table 10. Randomly generated PLAs – comparison of the result quality)

p / n	50	100	150	200
50	110/41/25 (58) 122/54/27/3.89	96/35/23 (90) 104/45/23/10.29	90/32/21 (147) 92/41/21/24.87	84/29/20 (199) 89/39/20/41.99
100	284/86/52 (46) 289/104/51/19.31	229/68/42 (94) 231/84/42/77.07	217/61/40 (140) 213/80/39/199.17	207/57/38 (140) 201/74/37/246.21
150	474/132/76 (43) 481/158/76/54.76	389/101/63 (101) 384/125/62/282.80	362/92/61 (116) 345/113/56/646.20	381/90/64 (64) 322/107/52/1066.14
200	678/177/101 (51) 686/209/101/162.62	553/137/83 (116) 539/165/81/730.91	492/125/75 (207) 480/149/72/1913.65	469/110/71 (277) 450/136/68/3372.66

Entry format: BOOM: # of literals / output cost / # of terms (# of iterations)
ESPRESSO: # of literals / output cost / # of terms / time in seconds

The second group of experiments was performed to compare the run-times. Again, the same randomly generated problems were solved, but this time BOOM was running until a solution of *the same or better quality* as ESPRESSO was reached. The quality criterion selected was again the sum of the number of literals and the output cost. The results given in Table 11 show that for all examples the same or better solution was found by BOOM in a shorter time than by ESPRESSO. Thus, even though iteration was employed, BOOM was faster than ESPRESSO. Moreover, the UCP problem was solved extra after each iteration in this experiment, in order to obtain a solution for comparison. Otherwise the stopping criterion couldn't be computed.

Table 11. Randomly generated PLAs – comparison of the run-time (same quality as ESPRESSO)

p / n	50	100	150	200
50	170/0,64 (12) 176/3,89	145/1,89 (21) 149/10,29	131/14,52 (73) 133/24,87	126/3,26 (25) 128/41,99
100	388/7,15 (23) 393/19,31	313/25,5 (48) 315/77,07	291/38,91 (56) 293/199,17	273/86,51 (83) 275/246,21
150	631/20,38 (25) 639/54,76	506/153,84 (70) 509/282,8	456/374,68 (105) 458/646,20	427/974,40 (161) 429/1066,14
200	890/71,97 (31) 895/162,62	697/467,63 (86) 704/730,91	625/1026,28 (149) 629/1913,65	582/1759,27 (220) 586/3372,66

Entry format: BOOM: # of literals + output cost / time in seconds (# of iterations)
ESPRESSO: # of literals + output cost / time in seconds

5.4.3 PRACTICAL PLAS

Results of some large practical examples that had to be minimized during a test pattern generator design for BIST [80], [81] will be shown in this subsection. In particular, the combinational logic transforming pseudo-random patterns into deterministic ones was designed.

In all the cases BOOM was run for 100 iterations. The *"Bench"* column indicates the name of the benchmark circuit. In the "*n/m/p"* column there are listed numbers of its inputs, outputs and care terms. Then, the results obtained by BOOM and ESPRESSO are shown. The resulting PLA complexity, in terms of the sum of the number of literals and the output cost, is given in in the next column ("*Size*"). The shadowed cells indicate cases where BOOM outperformed ESPRESSO, both for the result quality and run-time.

Table 12. Output Decoder design examples

		BOOM		ESPRESSO	
Bench	*n/m/p*	*Size*	*Time [s]*	*Size*	*Time [s]*
d_c1355 (1)	41/18/13	63	0.69	74	0.19
d_c1355 (2)	41/21/14	70	0.74	85	0.26
d_c1908	33/3/29	34	0.49	36	0.12
d_c2670 (1)	233/32/60	226	165.95	626	4,838.62
d_c2670 (2)	233/31/52	122	159.06	520	2,329.44
d_c2670 (3)	233/36/104	319	740.18	689	24,710.07
d_c7552 (1)	207/48/81	393	807.84	746	27,574.93
d_c7552 (2)	207/72/207	779	23,933.46	-	> 24 h
d_s420.1 (1)	34/6/42	44	0.75	52	1.58
d_s420.1 (2)	34/5/33	39	0.75	49	0.95
d_s838 (1)	67/24/61	71	3.15	117	27.94
d_s838 (2)	67/15/46	58	1.65	88	14.94
d_s953 (1)	45/2/25	9	0.13	14	0.11
d_s953 (2)	45/4/45	21	0.42	21	0.16
d_s1196	32/4/48	59	2.12	74	1.04
d_s1238 (1)	32/5/60	93	9.71	137	3.15
d_s1238 (2)	32/4/58	57	5.23	73	0.53
d_s5378 (1)	214/3/36	24	2.81	31	6.58
d_s5378 (2)	214/2/22	14	0.66	14	1.70
d_s9234 (1)	247/77/216	1310	18,835.60	-	> 24 h

		BOOM		ESPRESSO	
Bench	*n/m/p*	*Size*	*Time [s]*	*Size*	*Time [s]*
d_s9234 (2)	247/38/99	373	266.78	505	17,298.00
d_s9234 (3)	247/23/52	129	29.09	216	659.25
d_s13207.1 (1)	700/8/96	177	93.65	191	1,251.00
d_s13207.1 (2)	700/58/197	587	1,550.25	633	190,038.74
d_ s15850.1 (1)	611/96/313	395	3,416.40	-	> 24 h
d_ s15850.1 (2)	611/48/180	157	516.30	240	37,818.65
d_s38417	1664/1454/520	1518	1,923.00	-	> 24 h
d_s38584.1 (1)	1664/464/307	316	321.90	-	> 24 h
d_s38584.1 (2)	1664/464/45	22	46.60	62	20,361.71
d_b04 (1)	77/9/37	49	1.75	57	4.19
d_b04 (2)	77/4/29	18	0.32	22	0.58
d_b05 (1)	35/7/33	57	2.94	98	0.85
d_b05 (2)	35/2/15	8	0.07	9	0.06
d_b07 (1)	50/5/41	23	2.01	24	3.43
d_b07 (2)	50/1/24	2	0.02	2	0.80
d_b12 (1)	126/11/128	190	118.14	236	379.93
d_b12 (2)	126/7/66	91	15.52	115	18.27

It can be seen that BOOM outperformed ESPRESSO in the result quality in all the cases and mostly in the run-time as well. In some more complex cases (hundreds or up to thousands inputs) ESPRESSO did not return a result in more than one day, thus the measurement was terminated.

5.5 SCALABILITY

All the algorithms used in BOOM have polynomial time complexity (no more than quadratic), with all the instance size measures (number of inputs, outputs, and care terms). An experimental evaluation will be shown here, to see real time complexities.

The average time needed to complete one pass of the algorithm for varying sizes of the input truth table (PLA) was measured. The truth tables were generated randomly, all the specified terms were minterms. Single-output functions were considered in this experiment. 10 instances of each size were solved and average times were computed.

Figure 27 shows the growth of the average run-time as a function of the number of input variables (20-300), for different numbers of minterms (20-300 as well). It can be seen that the time complexity is almost linear.

The fact that the time complexity grows linearly with the number of input variables (while keeping the number of defined terms) expresses the main advantage of the BOOM algorithm. As the size of the Boolean space of the function grows exponentially with the number of input variables, the time complexity of most of the common minimization algorithms (ESPRESSO) grows exponentially too. In BOOM there is no chance for an exponential time grow, as there are no algorithms with an exponential complexity used. This allows BOOM to very efficiently minimize functions with an extremely large number of inputs.

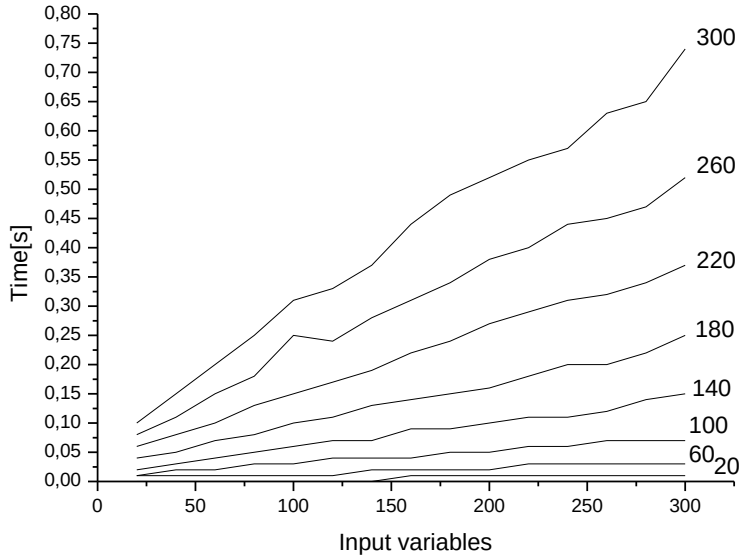


Figure 27. Time complexity (1)

The run-time dependency on the number of specified terms is shown in Figure 28. Again, the number of input variables was varied from 20 to 300 and figures for different numbers of specified minterms (20-300) were measured. We see that BOOM run-times grow more than linearly with the number of terms, which was expected. Indeed, the dependencies confirm the theoretical $O(p^2)$ complexity.

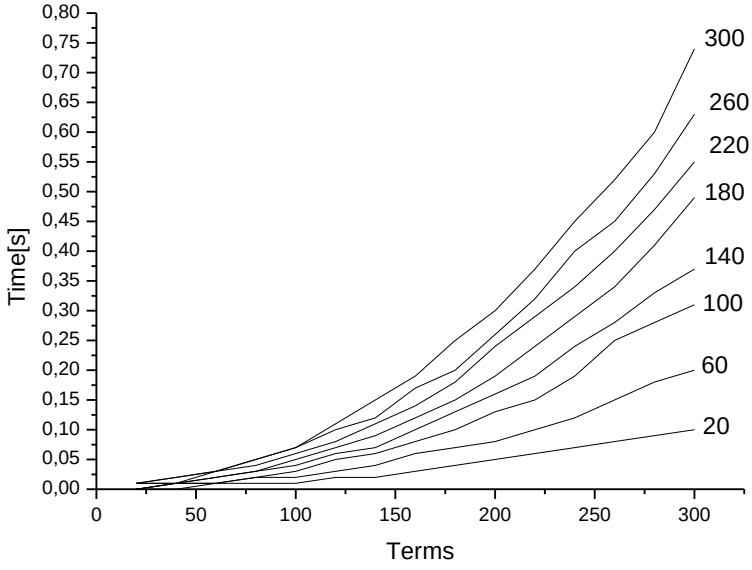


Figure 28. Time complexity (2)

5.6 SOLUTIONS ANALYSIS

As BOOM is randomized, different solutions are obtained from its different runs. As well as in Subsection 2.3, the counts of different solutions will be investigated, more thoroughly in this case.

The following experiment was performed: a subset of MCNC benchmarks [66] was processed, BOOM was run for 200 iterations (to ensure very quality results), 20% CD-Search mutations [83] (to increase the variety of solutions), repetitively 100-times for each circuit. All different solutions ever obtained (even in course of the iteration) were recorded. Note that all the results were prime and irredundant covers.

The detailed solutions analysis is shown in Table 13. For each benchmark, the total number of different solutions and different solutions with the best quality (number of literals here) are shown. Then, numbers and percentages of solutions, whose quality is less than 5% (10%, 20%, respectively) worse than the best solution are shown.

We can observe that for most of the circuits only one "best" solution was obtained, which was probably the optimum one. Of course, symmetric circuits, like *max512*, *sym10*, *Z9sym* [66] have adequate numbers of P-equivalent solutions [87].

However, a plentiful of different near-optimum solutions can be observed. This is illustrated in Figure 29 for the *ex1010* circuit. Such a behavior can be observed for most of the tested circuits.

Table 13. Different solutions counts for BOOM

benchmark	*solutions*	*best solutions*	≤ 5%	≤ 10%	≤ 20%
add6	2598	1	19 (1%)	218 (8%)	2580 (99%)
alu2	114	1	38 (33%)	73 (64%)	105 (92%)
alu3	1444	1	204 (14%)	495 (34%)	1133 (78%)
amd	2	1	1 (50%)	1 (50%)	1 (50%)
b12	212	1	26 (12%)	88 (42%)	212 (100%)
b9	929	1	52 (6%)	154 (17%)	319 (34%)
bench	4283	50	2599 (61%)	3803 (89%)	4213 (98%)
co14	1	1	1 (100%)	1 (100%)	1 (100%)
dc1	1	1	1 (100%)	1 (100%)	1 (100%)
dc2	16	1	1 (6%)	7 (44%)	16 (100%)
ex1010	17020	1	293 (2%)	6144 (36%)	16172 (95%)
ex7	909	2	47 (5%)	143 (16%)	315 (35%)
exep	2343	2	2139 (91%)	2334 (100%)	2343 (100%)
f51m	315	1	19 (6%)	70 (22%)	296 (94%)
ibm	1242	1	45 (4%)	169 (14%)	700 (56%)
in7	43	4	4 (9%)	18 (42%)	34 (79%)
inc	24	2	8 (33%)	11 (46%)	24 (100%)
jbp	1166	1	97 (8%)	429 (37%)	829 (71%)
life	1	1	1 (100%)	1 (100%)	1 (100%)
log8mod	37	2	19 (51%)	30 (81%)	37 (100%)
luc	4	2	4 (100%)	4 (100%)	4 (100%)
m1	9	1	2 (22%)	2 (22%)	9 (100%)
m2	141	1	87 (62%)	135 (96%)	141 (100%)
max512	1131	122	682 (60%)	947 (84%)	1129 (100%)
misj	15	1	1 (7%)	1 (7%)	1 (7%)
mlp4	338	2	25 (7%)	138 (41%)	328 (97%)
newcwp	2	1	1 (50%)	1 (50%)	2 (100%)
newtpla2	25	1	4 (16%)	7 (28%)	9 (36%)
p3	20	2	2 (10%)	2 (10%)	18 (90%)

benchmark	*solutions*	*best solutions*	≤ 5%	≤ 10%	≤ 20%
p82	12	1	6 (50%)	9 (75%)	12 (100%)
radd	344	1	3 (1%)	11 (3%)	80 (23%)
rckl	214	1	214 (100%)	214 (100%)	214 (100%)
rd73	1	1	1 (100%)	1 (100%)	1 (100%)
risc	1	1	1 (100%)	1 (100%)	1 (100%)
root	1163	3	249 (21%)	902 (78%)	1163 (100%)
ryy6	8499	1	566 (7%)	2649 (31%)	7406 (87%)
shift	414	1	230 (56%)	313 (76%)	414 (100%)
soar	1043	1	9 (1%)	59 (6%)	506 (49%)
sqn	113	1	23 (20%)	93 (82%)	113 (100%)
sym10	875	100	147 (17%)	242 (28%)	483 (55%)
t1	8	1	1 (13%)	1 (13%)	7 (88%)
test1	2937	1	1463 (50%)	2449 (83%)	2864 (98%)
test4	9702	4	58 (1%)	1116 (12%)	9666 (100%)
vg2	2	1	1 (50%)	1 (50%)	2 (100%)
vtx1	1149	4	929 (81%)	1009 (88%)	1103 (96%)
x1dn	8	1	1 (13%)	1 (13%)	1 (13%)
x2dn	5	1	1 (20%)	4 (80%)	5 (100%)
x9dn	741	1	634 (86%)	675 (91%)	731 (99%)
z4	395	1	37 (9%)	202 (51%)	381 (96%)
Z5xp1	153	1	61 (40%)	137 (90%)	153 (100%)
Z9sym	3178	99	751 (24%)	1329 (42%)	2541 (80%)

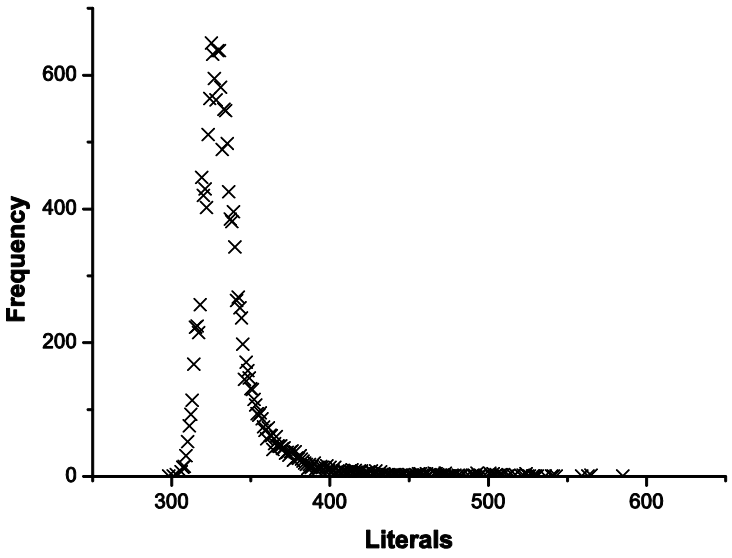


Figure 29. Distribution of solution qualities for *ex1010*

5.7 RANDOM NUMBER GENERATOR GRANULARITY EFFECTS

The influence of the random number generator granularity in the minimization process is illustrated in Figure 30. Here BOOM was partially derandomized, in the way described in Subsection 4.5 and the progress of the implicants number growth was traced. The same

function as in Section 5.3 was minimized (randomly generated, 20 input variables, 5 output variables, 200 care terms of average dimension 2).

The final result quality obtained after 1,000 iterations for different *RF*s is shown in Table 14 and the progress of the result quality during the 1,000 iterations is visualized by Figure 31. The values were obtained by averaging 20 BOOM runs (for each *RF* value).

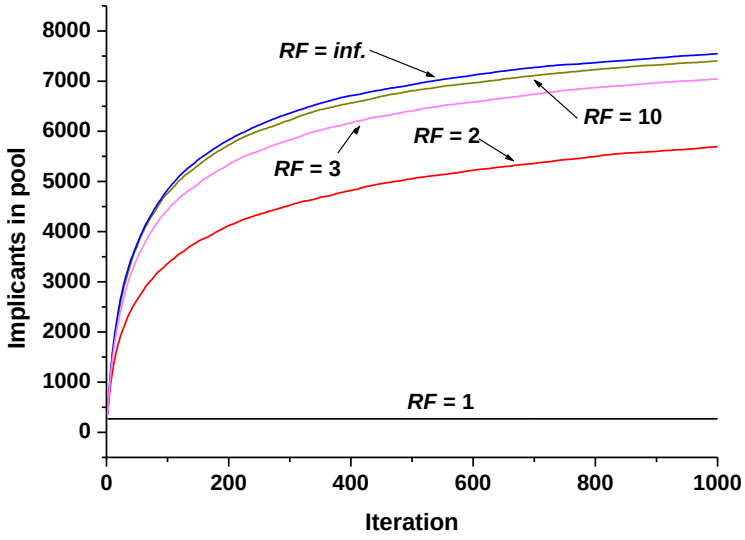


Figure 30. Derandomized BOOM – implicants number growth

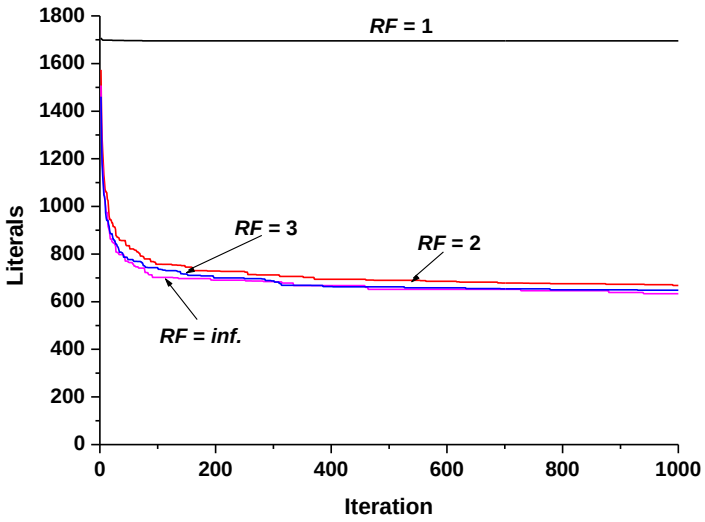


Figure 31. Derandomized BOOM – result quality

Table 14. 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 does not work 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 CD-Search choices (for our example circuit, fully randomized algorithm run, and 200 iterations) are shown in Figure 32. In 40% of cases there are no options. There are 2 options in less than 20% of cases, and the distribution curve sinks exponentially. The average number of choices was 3.35.

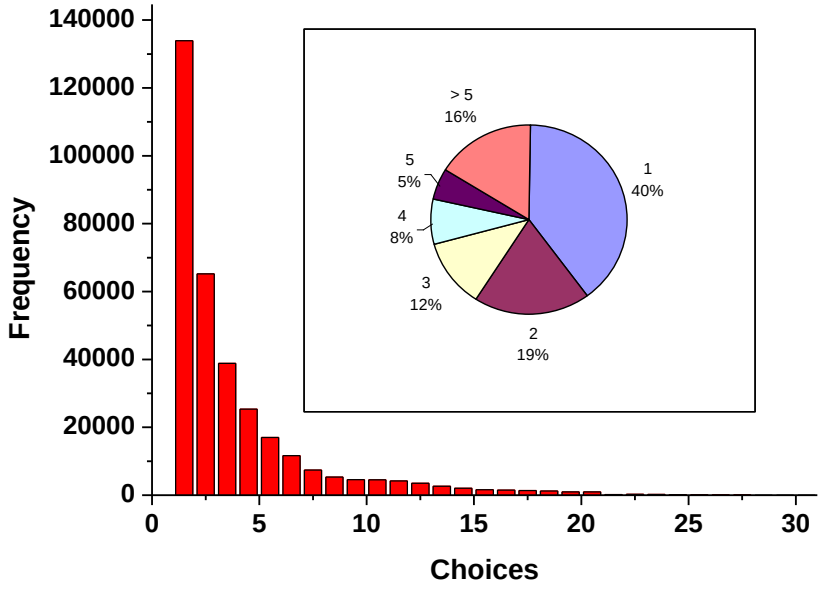


Figure 32. Numbers of choices

From the 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 the example was 31 only.

The number of possible random decisions in the subsequent phases (implicant expansion and reduction) is bounded by the number of variables. Only the number of decisions in the UCP phase is not limited; it grows with the number of processed implicants.

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. However, it definitely cannot be completely determinized.

5.8 Asymptotic Completeness of the Algorithm

In general, the overall BOOM algorithm (search strategy) can be declared as *asymptotically complete*, mostly because of mutations used in the CD-Search phase. Any implicant can be generated by application of mutations and if an exact covering problem is used [85], [86], also any optimum solution can be obtained.

Completeness, in sense of ability of obtaining *any* valid solution, is discussable. Nevertheless, it relies on the covering problem solving algorithm only. Therefore, any desired solution can be obtained, if the algorithm is modified properly. Of course, the solution must consist of prime

implicants only. If this condition is undesirable, a kind of relaxation must be applied in the implicants generation process.

5.9 CONCLUSIONS

Another application of randomized high-level iterative algorithms was shown here – the SOP minimizer BOOM. It starts with on- and off-set sets descriptions of a function and produces a minimized SOP form covering the on-set and not intersecting the off-set.

BOOM was found to be well scalable, its run-times do not grow significantly with the instance sizes and generally they are relatively small. Therefore, BOOM can be efficiently iterated, without a significant run-time increase.

When iterated, a valid solution can be obtained in every BOOM pass (iteration), if the covering problem is solved at the end of each iteration.

Because of the randomized nature of the algorithm, the solutions may differ. Moreover, even better solutions can be obtained by *combining* the generated solutions and solving the covering problem.

It was shown that BOOM is able to outperform the state-of-the-art SOP minimization tool ESPRESSO, mostly because of iteration – even though inferior results are usually obtained in the first iteration for complex problems, better solutions than ESPRESSO produces are obtained after several iterations of BOOM, moreover usually in a shorter time – even though iteration is used.

As in the previous algorithm, randomness allows obtaining many *different* solutions. This feature was exploited, e.g., in [88].

It was shown that both randomness and iteration are necessary for a successful BOOM run, at least for non-trivial problems, where the optimum solution is not already generated in the first iteration. However, partial derandomization affects BOOM only slightly; very few random numbers are required to make BOOM perform well.

6 FC-MIN – A SOP MINIMIZER

The second iterative randomized two-level minimizer discussed will be FC-Min [89], [90]. Its novelty and uniqueness consist in a special solution generation process. In contrast to other approaches, FC-Min finds a *rectangle cover* of '1's in the output part of the truth table first, and then group implicants are subsequently derived from this cover. No prime implicants of single functions are computed; only the necessary implicants needed to cover the whole on-set are produced.

As group implicants are highly important especially for functions with many outputs, this makes FC-Min superior to other minimizers for such problems. On the other hand, FC-Min is not suitable for problems with a small number of output variables; for a single-output function, as an extreme case, the cover is generated purely at random. Here the FC-Min algorithm cannot outperform the others (ESPRESSO, BOOM).

Randomness is employed in FC-Min in a special way too. In all the above-mentioned approaches (Sections 3 - 5) randomness was used when more than one equally valued choice were available. Conversely, randomness is used in FC-Min to ensure its *probabilistic* execution. The mentioned rectangle cover (which is an NP-hard problem [37] too) is generated greedily and almost ad-hoc, whereas optimality of its solutions is controlled by a random factor. Indeed, highly sub-optimal solutions must be often submitted to the subsequent phase, since optimality of the rectangle cover is in contrast with its feasibility for the next phase. Details will be given in Subsection 6.3.

Iteration is in FC-Min used both at low and high level. The low-level iteration is employed in the probabilistic Find Cover phase (see Subsection 6.2.1); high-level iteration can be used in the same way as in BOOM: different implicants obtained in each iteration are cumulated and the covering problem is solved at the end.

6.1 PRELIMINARIES

The general notation will be retained from the previous Section. However, a notation of input and output matrices must be introduced here, for simplicity of the further explanations.

Let us have a multi-output Boolean function of m output variables of n input variables. This function will be denoted as a *source function*. The input variables will be denoted as x_i, $0 \le i < n$, the output variables as y_j, $0 \le j < m$. Such a multi-output function will be described by a sum-of-products (SOP) form, particularly by a PLA [6]. The m output values of the care terms (both minterms and terms of higher dimensions may be used) are defined by a truth table. To the minterms that are not present in the truth table are implicitly assigned don't care values. I.e., the multi-output function is specified by its on-set and off-set as in the BOOM case. Don't cares can also be specified in the truth table explicitly.

The part of the truth table representing the terms (*cubes*) will be denoted as an *input matrix* **I**, the rows of the input matrix will be denoted as *input vectors*. The part defining output values of the terms will be called an *output matrix* **O***;* similarly, the rows of this matrix as *output vectors*. Each row of the output matrix defines values of the output variables for the values of input variables specified by the corresponding row in the input matrix. Two notations for the **I** matrix rows will be used interchangeably in the text:

1. an n-dimensional binary vector describing values of respective input variables,
2. an **I** matrix term (cube) describing a set of literals.

The number of **I** matrix columns corresponds to the number of input variables n, the number of **O** matrix columns is equal to the number of output variables m, the number of **I** and **O** matrix rows will be denoted as p. The **I** matrix may contain values '1' (input variable in its positive phase), '0' (input variable complemented), and '-' (input variable not present). The **O** matrix may

contain values ‘0’ (function’s output value 0), ‘1’ (function’s output value 1), and ‘-‘ (output don’t care – any output value is allowed).

The SOP minimization task is the same as in BOOM (see Subsection 5.1): to find the minimum cover of the on-set. The optimality may be measured as the number of product terms in the solution, the number of literals, the output cost, or their combination. Terms can be shared between more output functions. Then the respective term is accounted only once.

6.2 FC-MIN PRINCIPLES

There are two major phases of this algorithm: the *Find Cover* phase (which gave name to FC-Min) and the *Find Implicants* phase. After that, the implicants should yet be expanded in order to reduce the total number of literals. The number of terms cannot be reduced any more after the Find Implicants phase. The individual phases will be described in the following Subsections.

6.2.1 FIND COVER PHASE

First, let’s state some basic definitions [3], [37].

Definition 1

A *rectangle (R, C)* of a matrix **M**, **M**[*i, j*] ∈ {‘0’, ‘1’, ’-‘}, is a subset of rows *R* and columns *C*, such that **M**[*i, j*] ≠ ‘0’ for each $i \in R$ and $j \in C$.

$$\forall i \in R, \forall j \in C : \mathbf{M}[i, j] \neq '0' \tag{1}$$

The *size* of a rectangle |*(R, C)*| is the number of ‘1’s covered by it. Note that DCs (’-‘) can be included in the rectangle, however they do not contribute to its size for the purpose of the algorithm.

■

Definition 2

A set of rectangles **R** = {(R^k, C^k)} forms a *rectangle cover* of a matrix **M**, if all ‘1’ **M** matrix values are included in (*covered* by) at least one rectangle in this set. The cover needs not be disjoint; one ‘1’ may be covered by more than one rectangle. More formally, for a matrix **M** of dimensions (*p, m*), it holds:

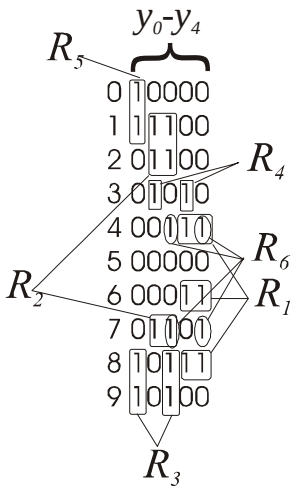
$$\forall i < p, \forall j < m : \mathbf{M}[i, j] = '1' \Rightarrow \exists k : i \in R^k \wedge j \in C^k \tag{2}$$

The *size* of a rectangle cover |**R**| is the cardinality of the set **R**, i.e., the number of rectangles.

■

Example

An example of a (minimum) rectangle cover is shown in Figure 33. for a binary matrix of 5 columns (let they be named y_0 - y_4) and 10 rows (numbered 0-9). The cover consists of six *rectangles*, R_1 – R_6. The sets of columns in the respective rectangles can also be represented by binary vectors, as depicted in Figure 33.



Rectangle (i)	R_i	C_i
R_1	{4, 6, 8}	$\{y_3, y_4\} \equiv 00011$
R_2	{1, 2, 7}	$\{y_1, y_2\} \equiv 01100$
R_3	{8, 9}	$\{y_0, y_2\} \equiv 10100$
R_4	{3}	$\{y_1, y_3\} \equiv 01010$
R_5	{0, 1}	$\{y_0, y_1\} \equiv 10000$
R_6	{4, 7}	$\{y_2, y_4\} \equiv 00101$

Figure 33. Example of a rectangle cover

■

In the *Find Cover* phase we find a rectangle cover of the on-set of the source multi-output function, i.e., the rectangle cover of the **O** matrix. By this we try to find potential implicants that could be included in the solution. The main idea will be described later.

Finding the optimum rectangle cover is an NP-hard problem [37], moreover the problem is constrained by the implicant validation procedure (see Subsection 6.2.2). For this reason, exact methods or commonly used heuristic algorithms cannot be efficiently used, hence a special *probabilistic* greedy heuristic was developed.

The heuristic is based on a gradual search for rectangles consisting of maximum '1's. First, the row containing most of yet uncovered '1's is selected as a basis for a rectangle. Then rows are being greedily appended to the rectangle, while the number of covered '1's increases, or at least does not decrease. The search may also be terminated at any time, with a given probability. For details see Subsection 6.3.

The algorithm producing one rectangle is described by the following pseudo-code:

```
FindRectangle(O) {      // O is the O matrix of dimensions (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); // enforced random termination
    return (R, C);
}
```

Figure 34. Find rectangle algorithm

Example

The construction of the first rectangle in our example (Figure 33) will be shown here. The row no. 8 with four '1's is selected as the basis. Then, the current rectangle is:

$$(R^1, C^1) = (\{8\}, \{y_0, y_2, y_3, y_4\})$$
$$|(R^1, C^1)| = 1 \times 4 = 4 \qquad (3)$$

Now we continue the search for a next row to add in order to increase the number of covered '1's. In the example, when the row no. 4 is added, the number of covered '1's increases to 6.

$$(R^1, C^1) = (\{4, 8\}, \{y_2, y_3, y_4\})$$
$$|(R^1, C^1)| = 2 \times 3 = 6 \quad (4)$$

Next, there are two equally valued candidate rows to add - no. 6 and no. 7. Addition of one of these rows to the rectangle would not increase the number of covered '1's, however it will not decrease it either. Let us assume the row no. 6 is chosen (if two or more equal choices happen, a random decision is made).

$$(R^1, C^1) = (\{4, 6, 8\}, \{y_3, y_4\})$$
$$|(R^1, C^1)| = 3 \times 2 = 6 \quad (5)$$

No further row additions non-decreasing the size of the rectangle are possible, thus the rectangle generation is terminated.

After finding one rectangle, the **O** matrix '1's that are covered by it are substituted by '-' values and we continue the search for other rectangles until all the '1's in the output matrix are covered. This basic algorithm is greedy; no backtracking is involved. Thus, the asymptotic complexity of the overall algorithm is $O(m.p^2)$, since at most p rectangles can be produced by this algorithm.

■

6.2.2 IMPLICANT GENERATION PHASE

Rectangles obtained in the Find Cover phase represent potential group implicants of the minimized multi-output function. The structure of the terms (meaning literals in the terms) is not known yet; the rectangles define just the sets of '1's the implicants must cover. However, the size of the rectangle cover directly determines the number of implicants in the final solution.

This second minimization step consists in finding implicants covering exactly the same sets of **O** matrix '1's as the rectangles do.

Theorem

Let **R** be a rectangle cover of the **O** matrix. The *minimum term* covering the set of '1's that a rectangle $(R^k, C^k) \in \mathbf{R}$ covers is constructed as a *minimum supercube* of all **I** matrix rows (terms) indexed by the set R^k. Here the minimum is understood in terms of its dimension (i.e., the cube has maximum of literals).

■

Proof

For any single element of R^k, $r_i \in R^k$, the source function term represented by the r_i-th **I** matrix row naturally covers all '1's covered by R^k, by definition of the truth table. Let this term be denoted t_i. A term $t_{i,j}$ covering '1's covered by two elements of R^k, r_i and $r_j \in R^k$, $i \neq j$, must include both t_i and t_j, i.e., $t_i \subseteq t_{i,j}$ and $t_j \subseteq t_{i,j}$. Therefore, $t_{i,j} = t_i \cup t_j$, i.e., it is a supercube of t_i and t_j. This can be inductively extended for all R^k elements.

■

Next, the term obtained from the intersection must not intersect the off-set of any function indexed by elements of C. If the minimum supercube does so, the rectangle must be invalidated, since no adequate term is possible to generate.

Example

Let us assume a 5-input and 5-output incompletely specified function defined by 10 minterms, in form of a truth table. The input variables are named x_0 - x_4, output variables y_0 - y_4. The rest out of the total 32 minterms are don't cares. The truth table is shown in Figure 35. The output matrix is the same as in Figure 33, therefore we will refer to the Find Cover phase solution (see Subsection 6.2.1).

	x_0-x_4	y_0-y_4
0	11010	10000
1	10000	11100
2	01001	01100
3	01111	01010
4	00110	00111
5	01110	00000
6	10110	00011
7	00001	01101
8	10101	10111
9	11100	10100

Figure 35. The FC-Min example function

The way of computation of the minimum implicants t_1 – t_6 derived from the rectangles R_1 - R_6 is shown in Figure 36. The respective **I** matrix rows are indicated in brackets.

$t_1 \leftarrow R_1$	$t_2 \leftarrow R_2$	$t_3 \leftarrow R_3$	$t_4 \leftarrow R_4$	$t_5 \leftarrow R_5$	$t_6 \leftarrow R_6$
(4) 00110 (6) 10110 (8) 10101 -01–	(1) 10000 (2) 01001 (7) 00001 --00-	(8) 10101 (9) 11100 1-10-	(3) 01111 01111	(0) 11010 (1) 10000 1-0-0	(4) 00110 (7) 00001 00---
($x_1'x_2$)	($x_2'x_3'$)	($x_0x_2x_3'$)	($x_0'x_1x_2x_3\,x_4$)	($x_0x_2'\,x_4'$)	($x_0'x_1'$)

Figure 36. Implicants generation

Since none of the generated terms intersects off-sets of functions they implicate, all of them are valid. The solution of the example is a PLA shown in Figure 37. The output matrix of the result is derived from the rectangles columns, see the table in Figure 33.

The case where invalid implicants are generated will be discussed in Subsection 6.2.4.

Note the difference of the PLA descriptions in Figure 35 and Figure 37. In Figure 35 terms not specified in the table were implicitly assigned as DCs. In Figure 37 the minimization result is shown, i.e., the table corresponds to the final PLA (SOP) implementation. Thus, minterms not specified in the table implicitly belong to the off-set.

```
t1: -01-- 00011
t2: --00- 01100
t3: 1-10- 10100
t4: 01111 01010
t5: 1-0-0 10000
t6: 00--- 00101
```

Figure 37. Implicant generation phase solution

■

Note that the asymptotic complexity of the implicant validation procedure is polynomial with the number of inputs - $O(n^2.p)$, since the off-set is explicitly specified.

6.2.3 IMPLICANT EXPANSION

The Implicant generation phase produces *minimum* implicants, i.e., a set of valid implicants covering all '1's in the output matrix and having maximum of literals. These implicants can be further expanded by removing literals (i.e., '0' and '1' values from the input matrix), as long as the expanded terms do not intersect the off-set of functions they implicate. Even though the validity of possible expansions is checked in a polynomial time (the offset is explicitly specified), there is an exponential number of sequences of expansions. Therefore, a simple heuristic is used. The terms are expanded by a "pecking out" method allowing us to make a balanced expansion of the terms: all literals in the matrix are tried for removal in a *random order*. If the expanded term does not intersect the off-set, the removal is kept permanent, otherwise the literal is not removed and the search continues until all literals are processed.

The expanded matrix, which is the final solution of the leading example, is shown in Figure 38. Literals that have been removed by the implicant expansion phase are shadowed.

t_1: -01-- 00011	$y_0 = x_2x_3' + x_0x_2'$
t_2: --00- 01100	$y_1 = x_2'x_3' + x_3x_4 + x_0x_2'$
t_3: --10- 10100	$y_2 = x_2'x_3' + x_2x_3' + x_0'x_1'$
t_4: ---11 01010	$y_3 = x_1'x_2 + x_3x_4$
t_5: 1-0-- 11000	$y_4 = x_1'x_2 + x_0'x_1'$
t_6: 00--- 00101	

Figure 38. FC-Min – the final solution

After the expansion is performed, it may happen that some terms become redundant, i.e., the on-set covered by them is already covered by other (those expanded) terms. Therefore, a covering problem is solved at the end, to obtain an irredundant cover. The same procedure as in Subsection 5.2.4 is used.

6.2.4 INCREMENTAL IMPLICANTS GENERATION

Until now it was assumed that each term created as the minimum supercube of **I** matrix terms is a valid implicant of all output functions it should implicate (these are directly determined by the implicant's respective rectangle rows). However, this is definitely not the rule. Since the Find Cover phase computes rectangles independently of the **I** matrix, it may happen that the generated implicants intersects the offset.

Example

Let us assume a possibly obtained rectangle in our leading example (Figure 33), R^7 = ({4, 8}, {y_2, y_3, y_4}). Actually, this is the rectangle obtained in the second step of generation of R^1. The minimum supercube of **I** matrix terms {4, 8} is (-01--) = $x_1'x_2$. This term must implicate functions y_2, y_3, and y_4. However it does not, since it intersects the off-set of y_2 (see Figure 35, row no. 6). For this reason, this term (or, better, the rectangle R^7) cannot be included into the solution.

```
(4) 00110
(8) 10101
    -01--
```
$x_1'x_2$

Figure 39. Example of an invalid implicant

■

If such a rectangle was produced, it would be impossible to find implicants for the computed rectangle cover. Therefore, the cover must be recomputed. One possibility is to split the cover in order to make supercubes of fewer terms. This approach leads to a rapid growth of the number of terms in the final solution.

Another possibility is to recompute the whole cover, thus repeat the phases until a valid solution is found. Such an approach causes a great growth of the run-time and also the algorithm often locks in an infinite loop.

It was found that the best way of solving this problem is an *incremental implicant generation*. In this approach the two main FC-Min phases are not separated; once one rectangle is generated, its respective minimum implicant is produced immediately and, if it is not valid, only the last rectangle is discarded and a different one is looked for.

This is the low-level iteration basis. The single implicant generation process is iterated, until a valid one is found. In each iteration implicants are generated in a randomized way, while their statistical properties can be influenced by a parameter, see the following Subsection.

The whole algorithm can be described by the following pseudo-code. The inputs of the algorithm are the input matrix **I** and the output matrix **O**, the output is a matrix **S**, representing the solution (PLA, SOP).

```
FC_Minimize(I, O) {
   S = ∅;
   do {
       do {
           (R, C) = FindRectangle(O);
           t = GenerateMinimumTerm(I, (R, C) );
       } while !IsValid(t, I);
       S = S ∪ t;
   } while !AllCovered();
   Expand(S);
   return S;
}
```

Figure 40. The FC-Min minimization algorithm

6.3 THE DEPTH FACTOR

Finding rectangles consisting of maximum of '1's is advantageous for the minimization, however the more rows the rectangle has, the smaller is the probability of validity of an implicant induced by it. Moreover, even the algorithm in Figure 40 may end up in an infinite loop, in case of repeated generation of one single non-valid rectangle. Thus, the greedy `FindRectangle` algorithm is driven by a *depth factor* (*DF*). Since the rectangles are produced row by row, after each row addition we may decide, whether to extend the cover to more rows, or to terminate the rectangle generation, even if it could grow yet bigger. The decision is made *at random* with a *probability* given by *DF*, see Figure 34. For instance, when *DF* = 0.5, there is 50% probability of the search termination after each row addition. For *DF* = 0, the search is terminated immediately after one row selection, thus a single-row rectangle is always produced. Here the FC-Min phase is downgraded to a mere gradual selection of all the **I** matrix rows into the solution, thus no minimization is performed in the Find Cover phase. Having a source function of *p* terms, the solution will consist of *p* implicants created just by expansion of the input terms.

For *DF* = 1, the search for rows continues until there are no candidates increasing the cover size. Hence, it is very likely that the algorithm will end up in an infinite loop, since rectangles having maximum of rows will be deterministically produced.

Note that for *DF* < 1, there is always a non-zero probability of producing a single-row rectangle. Such a rectangle *always* induces a valid implicant (which is equal to the respective **I** matrix row). Therefore, the algorithm can never get stuck in an infinite loop.

Now it becomes apparent why the algorithm does not perform well for single-output functions. Imagine a single-output function and *DF* = 1. Then the cover will consist of one rectangle spanning over all rows. The implicant induced by this rectangle apparently cannot be valid (unless the function is a tautology). For smaller *DF*s, the solution will be produced. However, note that when there is more candidate rows in the `FindRectangle` function (see Figure 34), one is

selected randomly. For single-output functions, *any* row increases the cost function exactly by one, thus all rows are equally valued candidates. Therefore, for smaller *DF*s, implicants are generated purely at random, independently of the source function terms. Therefore, good solutions can be hardly expected.

Since the depth factor significantly influences the generation of the cover, the choice of the *DF* value is crucial for reaching desirable solutions in a reasonable time. There are two aspects that *DF* influences:

- Higher values of *DF* force the algorithm to generate "deeper" terms, i.e., terms that span over many source on-set terms. These terms are consequently implicants of less output variables, since the cardinality of *R* gradually decreases in the rectangle generation process (see Figure 34). These rectangles are often not valid, thus they have to be frequently recomputed in the process (see Figure 40). This involves a rapid increase of the Find Cover phase run-time.
- Higher values of *DF* also induce producing less terms in the Find Cover phase. This means, the final solution may be possibly consisted of less terms, and definitely less terms will be processed in the Implicant Expansion, Covering problem solution and also in the overall minimization process, if high-level iteration (see Subsection 6.4) is applied. This could, conversely, reduce the total minimization run-time.

Therefore, a thorough analysis of the overall algorithm behavior is required, to select the "optimum" value of *DF*.

The ratio of the total number of rectangle computations (trials) to the number of valid rectangles (hits) as a function of *DF* is shown in Figure 41, for a randomly generated PLA of 50 inputs, 10 outputs and 2,000 defined on-set minterms. A random function was selected so to suppress any possible structural singularities appearing in standard benchmark circuits. However, the conclusions presented in this subsection can be generalized for any circuit.

It can be seen that the ratio grows hyperexponentially with *DF* (notice the logarithmic y-axis), and so does the Find Cover phase run-time, since it grows linearly with the number of rectangle computations.

Figure 42 shows the numbers of terms obtained in the Find Cover phase (upper curve) and terms in the final solution (lower curve). The curves sink linearly with increasing *DF*, while the maximum reduction of 32% was obtained in this case (the original function had 2,000 terms). Notice here that the Find Cover phase basically determines the number of terms in the final solution; very slight improvement in the terms number is obtained by the Implicant Expansion and Covering problem solution phases.

However, this is not the case of the literals number. A similar graph, now displaying the numbers of literals, is shown in Figure 43. The minimization effect of the Find Cover phase can be seen clearly. For $DF = 0$, all the minimization effort is left for the Implicant Expansion phase (notice that the original function had 100,000 literals). For large DFs, the Find Cover phase does a big deal of the job, so the Implicant expansion improves the result quality only slightly.

The most important consequence of the above-mentioned observations is the influence of *DF* on the overall minimization run-time. Since the Implicant expansion phase is relatively time-consuming, the less implicants are expanded, the better. The Find Cover phase and the overall run-times for varying *DF* are shown in Figure 44. Even though the Find Cover phase run time grows hyperexponentially with *DF*, there is an apparent minimum in the overall run-time between $DF = 0.85 – 0.95$. Therefore, the trade-off between the run-time and solution quality can be found in this *DF* region. If high minimization effort is required, the *DF* may be increased, at expense of a rapid run-time growth.

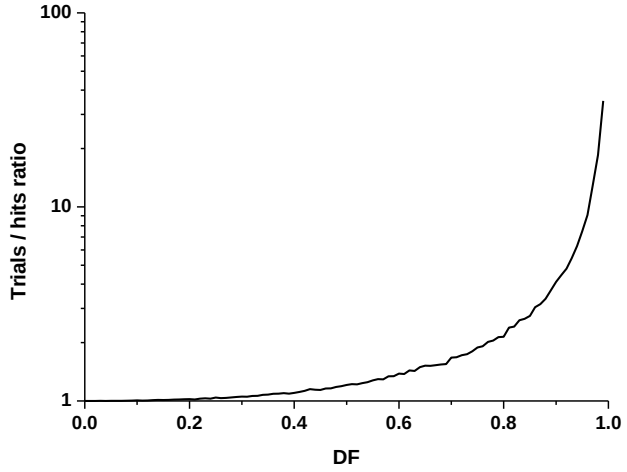


Figure 41. Ratio of the trials to the hits

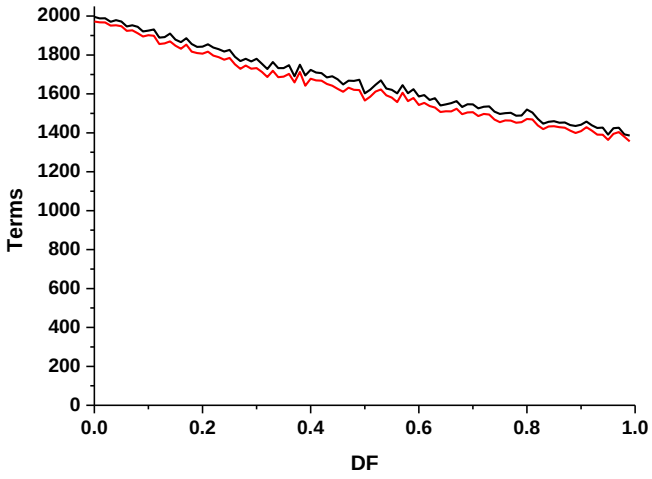


Figure 42. Terms obtained after Find Cover (upper curve) and terms in the final solution (lower curve)

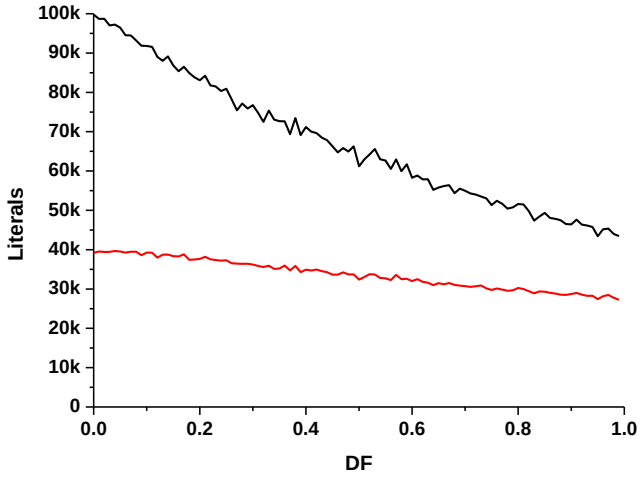


Figure 43. Literals obtained after Find Cover (upper curve) and in the final solution (lower curve)

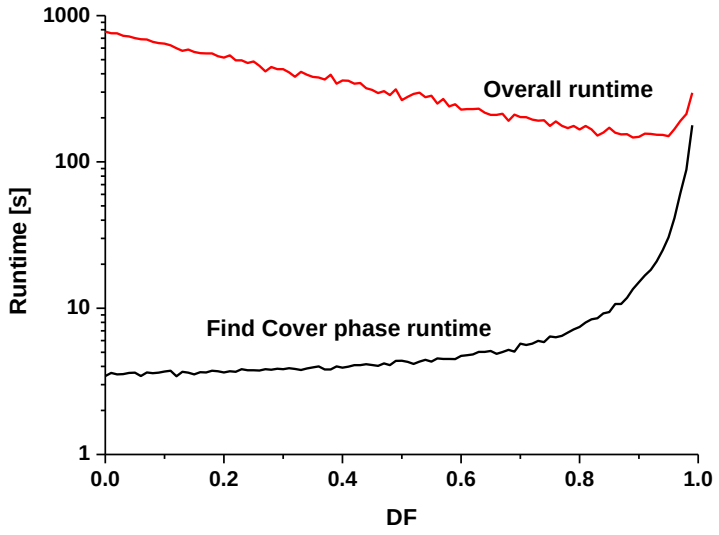


Figure 44. Minimization run-time

6.4 HIGH-LEVEL ITERATION OF FC-MIN

Since all the FC-Min phases are randomized, high-level iteration can be used in the same way as in BOOM (see Section 5), with the same effects. Particularly, Find Cover is the phase producing implicants, which are stored in the implicant pool. Then they are expanded (by expansion methods described in Subsections 5.2.2 and 6.2.3). This process is iterated and the covering problem is solved at the end.

The influence of the depth factor (*DF*) on the run-time is worth studying here.

Figure 45 illustrates the influence of the depth factor *DF* on the implicants growth rate. The sample problem solved was a randomly generated function of 20 input and 5 output variables, with 200 terms defined. The average dimension of the terms was 2.

When increasing *DF*, many different implicants are generated in each iteration step, allowing a faster implicants growth. However, more implicants involve a more time consuming covering problem solution phase. Therefore, a trade-off between the run-time and result quality must be found again.

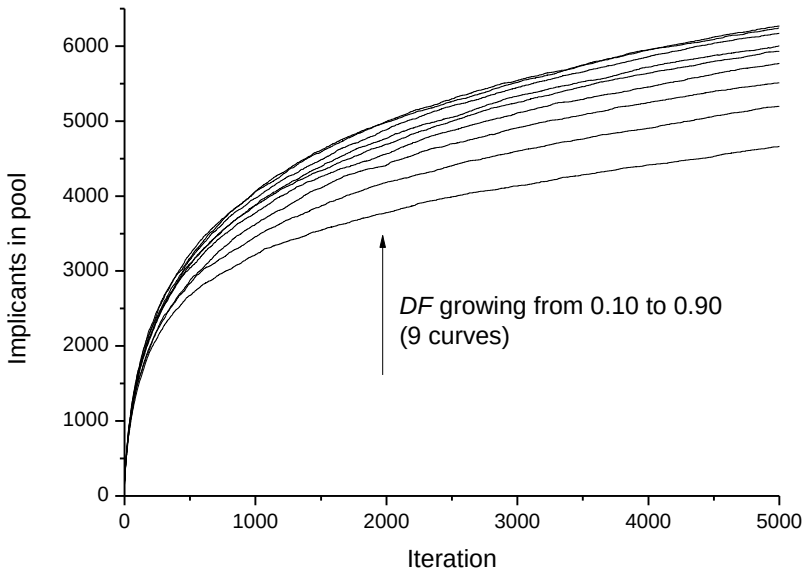


Figure 45. Influence of *DF* on the implicant growth rate

6.5 COMPARISON WITH ESPRESSO AND BOOM

The performance of FC-Min will be evaluated here, in comparison with ESPRESSO [6] and BOOM [75], [76]. Standard benchmarks and randomly generated problems were processed.

All the BOOM experiments were run on a standard PC with a 900 MHz Athlon processor and 256 MB of RAM.

6.5.1 MCNC BENCHMARKS

As well as BOOM, the FC-Min algorithm was tested on standard MCNC benchmarks [66] and compared the results and run-times with ESPRESSO [6].

Table 15 shows the results of the suggested MCNC benchmarks [66] and those where FC-Min has reached a better result than ESPRESSO (the bottom part of the table). The column *n/m/p* describes the numbers of input and output variables and the number of defined terms of the particular benchmark. The ESPRESSO and FC-Min columns indicate the run-times in seconds and the numbers of literals of the resulting SOP forms, the output costs (the number of inputs into all output OR gates), and the numbers of product terms. The shadowed cells indicate the shorter FC-Min run-time or equal or better result quality obtained, respectively.

The depth factor set to 0.5 and only one iteration were sufficient to obtain satisfactory results for these circuits.

There were 120 benchmark problems solved, plus 19 so-called “hard” MCNC benchmarks. As a result, 86 (72%) of them were solved by FC-Min in a shorter time than by ESPRESSO. In 103 cases (86%) FC-Min reached the same or better result (better results, in terms of the number of literals were obtained in 8 cases) and in 80 cases (67%) the same or better result was reached in a shorter time than by ESPRESSO.

Table 15. FC-Min - MCNC benchmarks

		ESPRESSO		FC-Min (*DF* = 0.5)	
benchmark	*n/m/p*	*time*	*lit/out/terms*	*time*	*lit/out/terms*
b12	15/9/72	0.08	149/59/42	0.01	148/58/43
cordic	23/2/2105	1.86	13825/914/914	8.08	13825/914/914
cps	24/109/855	0.33	1890/946/163	1.30	1890/946/163
duke2	22/29/404	0.09	751/245/86	0.14	751/245/86
ex1010	10/10/1304	0.50	1974/746/284	0.44	1976/742/284
ex4	128/28/654	0.62	1649/279/279	2.98	1649/279/279
misex2	25/18/101	0.07	183/30/28	0.01	183/30/28
misex3c	14/14/1566	0.98	1306/253/197	0.61	1306/255/197
pdc	16/40/822	0.83	828/432/136	0.32	912/520/145
rd84	8/4/511	0.12	1774/296/255	0.15	1774/296/255
spla	16/46/837	0.71	2558/643/251	0.84	2648/749/260
alu4	14/8/1184	0.59	4445/644/575	1.49	4443/644/575
clip	9/5/271	0.10	630/162/120	0.05	621/162/120
dc2	8/7/101	0.05	207/52/39	0.01	206/51/39
in4	32/20/603	0.17	2151/411/212	0.61	2145/411/212
m4	8/16/329	0.16	640/518/105	0.06	640/509/105
newxcpla1	9/23/93	0.07	197/86/39	0.01	196/86/39
opa	17/69/382	0.11	559/540/79	0.17	560/524/79
soar	83/94/779	0.94	2454/549/353	8.01	2445/549/353
x6dn	39/5/310	0.08	641/177/82	0.04	640/177/82

6.5.2 RANDOMLY GENERATED PROBLEMS – ONE ITERATION

The second set of problems on which FC-Min was tested were randomly generated functions, functions with no special properties (no aggregated ones in the output matrix, etc.) With a help of such problems we can easily observe the properties and scalability of the algorithm. As well as for BOOM, one of the reasons why FC-Min was developed was a need to synthesize the combinational logic for built-in self-test, namely the output decoder transforming the LFSR patterns into test patterns pre-generated by an ATPG tool. Both the LFSR and ATPG patterns mostly have a random nature, and thus the randomly generated benchmarks simulate these practical problems very well [80], [81].

Problems with a varying number of input variables and terms were generated, the number of outputs was fixed to 15. These artificial benchmarks were solved by FC-Min, BOOM and ESPRESSO to compare the performance. Here only one iteration of BOOM and FC-Min was performed, the FC-Min depth factor was set to 0.9.

The results of the minimization are shown in Table 16. The number of inputs increases in the horizontal direction (*n*), the number of care terms in the vertical direction (*p*). Each of the cells contains average values of ten problems of the same size that were solved, to ensure steady statistical values. The first row of each cell in the table contains results obtained by ESPRESSO, the second one the result obtained by BOOM and the third one by FC-Min. The first number in each line indicates the run-time in seconds, the second one the number of literals in the SOP form, the output cost follows and the last value indicates the number of product terms.

We can see that in all the cases FC-Min completed the minimization in a significantly shorter time than ESPRESSO and BOOM, while the result quality is comparable.

Table 16. Randomly generated problems – one iteration

p / n	25	50	100	150
50	2.27/232/341/49 1.30/413/220/87 0.27/315/305/59	11.34/219/318/48 1.40/428/156/94 0.29/304/241/58	46.35/202/301/46 1.59/412/121/90 0.34/293/195/56	94.64/203/303/47 1.73/371/96/84 0.41/283/181/54
100	10.22/577/687/98 2.59/998/506/168 0.50/716/648/110	100.14/537/576/91 2.56/1103/342/190 0.51/718/491/113	369.45/510/569/90 3.02/1050/244/186 0.64/676/413/106	883.24/488/554/88 3.53/943/186/168 0.79/647/372/102
125	14.44/772/849/123 3.51/1333/650/211 0.66/952/846/137	148.96/710/728/114 3.49/1468/449/237 0.65/927/642/137	756.21/666/704/110 4.23/1408/317/231 0.86/880/519/131	2146.03/652/674/108 4.82/1252/243/211 1.09/829/473/124
150	23.35/973/1005/147 4.71/1691/785/255 0.86/1182/1007/163	283.88/892/869/136 4.63/1849/563/285 0.84/1164/779/164	1111.23/833/800/129 4.72/1761/378/278 1.04/1098/638/157	3422.94/798/773/126 5.65/1613/295/256 1.33/1039/573/148

Entry format: time [s] / # of literals / output cost / # of terms

Lines: ESPRESSO, BOOM, FC-Min

6.6 RANDOMLY GENERATED PROBLEMS – SAME TIME

Next, the same set of problems was solved, but taking advantage of the high level iteration this time. The functions were minimized by ESPRESSO first, and then both by BOOM and FC-Min, while the run-time was set to meet the run-time that ESPRESSO needed to reach a solution.

The results are shown in Table 17. The format of the table is retained from the previous example, except of that only the ESPRESSO run-time is shown, while the number of iterations (to meet that time) is given in the parentheses for BOOM and FC-Min.

Here FC-Min gives much better results than ESPRESSO, especially for problems with many input variables. For most of these problems FC-Min outperformed BOOM as well, due to a relatively high number of outputs (15).

However, for problems with a low number of output variables BOOM is faster and the result quality is better too. Thus, for an efficient minimization we have to decide whether to use BOOM or FC-Min, judging by the number of outputs.

Table 17. Randomly generated problems – same time as ESPRESSO

p / n	25	50	100
50	2.15/233/346/49 340/246/70(2) 290/264/58(8)	10.80/218/324/48 294/189/61(7) 252/185/50(28)	51.96/204/309/47 247/139/53(27) 214/150/43(81)
75	5.62/400/513/74 525/381/95(3) 465/394/83(13)	34.37/370/463/70 466/276/86(12) 404/279/71(47)	154.71/357/438/68 423/218/79(35) 357/223/62(99)
100	11.24/581/673/99 768/528/127(4) 659/543/110(19)	84.48/546/586/92 665/358/111(16) 571/365/92(63)	416.29/520/564/90 600/287/102(44) 498/301/80(118)
125	17.75/773/845/123 1010/616/160(4) 868/674/138(22)	157.19/706/722/113 872/459/137(17) 745/456/115(71)	895.25/657/700/110 765/359/122(52) 650/374/99(137)

Entry format: ESPRESSO: time [s] / # of literals / output cost / # of implicants
next lines: # of literals / output cost / # of implicants (iterations)
Lines: ESPRESSO, BOOM, FC-Min

6.7 INFLUENCE OF THE NUMBER OF OUTPUTS

As it was stated above, FC-Min is efficient (in terms of both the run-time and result quality) for functions with many output variables. Conversely, its quality lacks for single-output functions, where the solution is generated purely at random. This issue will be studied here experimentally.

Randomly generated functions with 50 input variables, 200 specified terms of the average dimension 0.5 were minimized both by FC-Min and ESPRESSO and the result qualities (number of solution terms) was compared. The ratio of these two values, as a function of output variables, is shown in Figure 46. Averages from 20 runs (20 different randomly generated functions for each output variables count) were computed.

We can see that for single-output functions FC-Min produces very inferior results, more than twice the size of ESPRESSO (ESPRESSO produced results 45% the size of the FC-Min results). This lack in quality decreases with the increase of the number of outputs (m), for $m > 5$ the ratio becomes steady. Let's note that FC-Min does not reach the quality of ESPRESSO, because only one FC-Min iteration was conducted, for purpose of credibility of the results. If FC-Min was run for more iterations, a variety of (almost random, in the single-output functions case) implicants would be generated and the minimization problem would be left to the covering problem solution phase. Much better results would be definitely obtained, but they would not precisely reflect the properties of FC-Min, or particularly, the Find Cover phase.

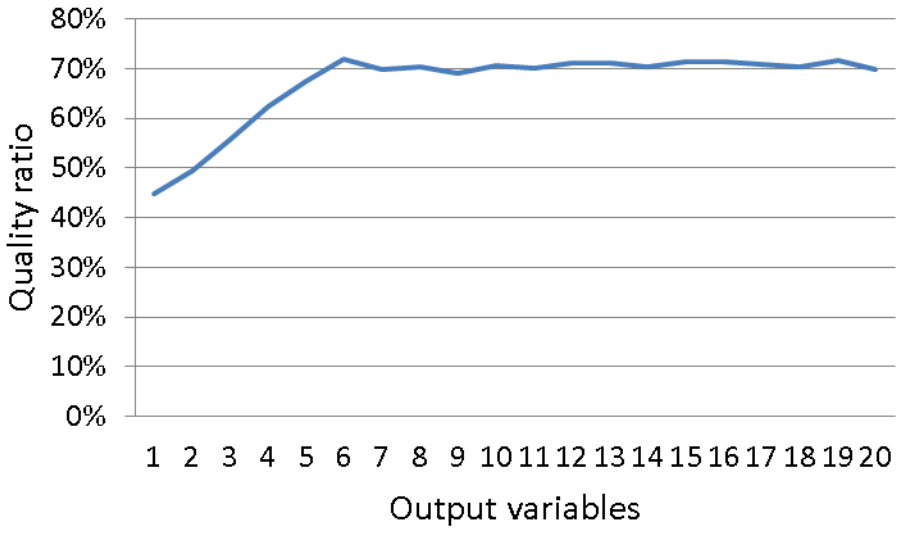


Figure 46. Influence of the number of output variables

6.8 FC-MIN SCALABILITY

Since FC-Min is a probabilistic randomized algorithm, it is difficult to determine its time complexity analytically. In order to estimate the time complexity of the method, FC-Min was run on a large number of randomly generated problems with one parameter varying each time, while the minimization times were recorded.

The following figures show the time dependencies on the number of input variables (Figure 47), output variables (Figure 48) and the number of care terms (Figure 49). Values of the fixed parameters are indicated in the figures, the depth factor was set to 0.9.

No exponential growth of time can be observed in any of the curves, thus the method can be scaled to very large problems while the run-time remains minimal.

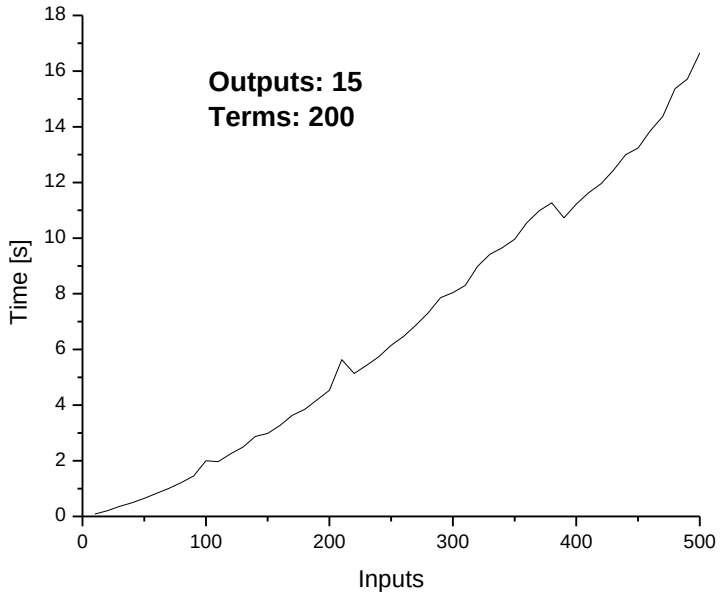


Figure 47. Time complexity as a function of the number of inputs

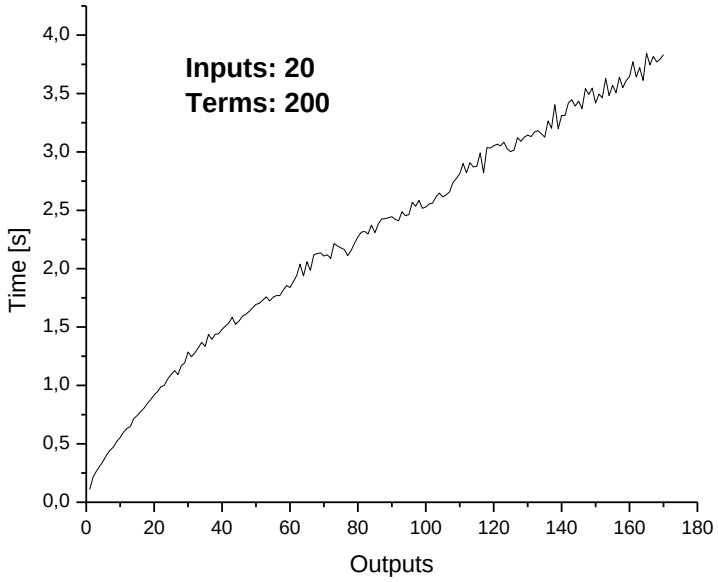


Figure 48. Time complexity as a function of the number of outputs

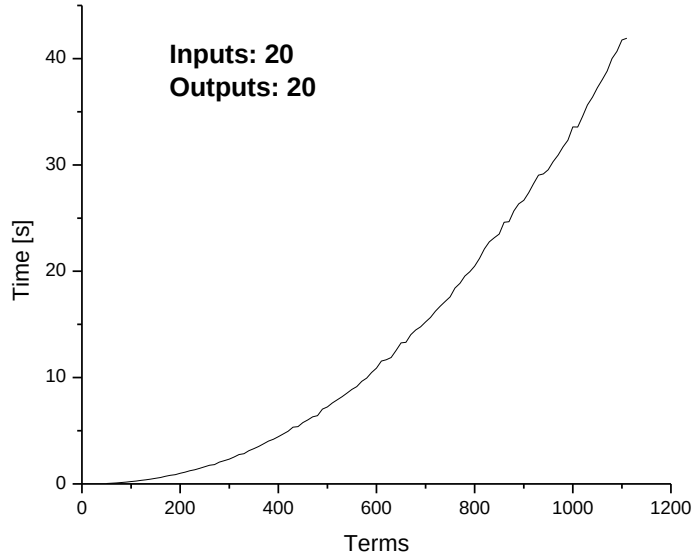


Figure 49. Time complexity as a function of the number of the care terms

6.9 RANDOM NUMBER GENERATOR GRANULARITY EFFECTS

Random nature of the Find Cover algorithm guarantees that the search will ever stop. Next, decreased randomness (random number generator granularity, see Subsection 4.5) decreases the variety of implicants generated by FC-Min. When FC-Min is run iteratively, this will involve a reduced implicants growth rate, as in the BOOM case. However, since the algorithm termination condition is *continuous* (`random()` generates real numbers here), much higher granularity is required for a successful algorithm run. This is documented by Figure 50. A randomly generated function having 20 inputs, 5 outputs, and 200 terms was minimized, *DF* was set to 0.8. The values were obtained by averaging 20 runs. The growth of the number of implicants during 1,000 iterations, for different randomness factors (*RF*s, see Subsection 4.5) 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 was expectable.

The solution quality is affected in the same way. *RF* of at least 1,000 is required, in order to approach the solution quality of the fully randomized process. The progress of the solution quality (number of literals) is depicted in Figure 51.

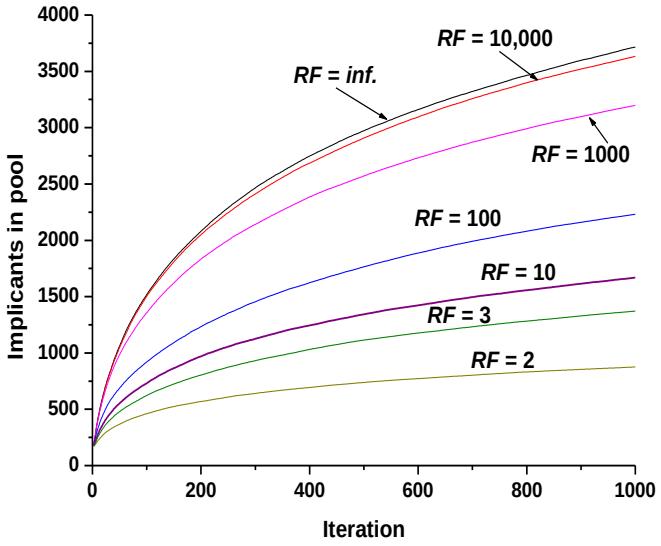


Figure 50. Derandomized FC-Min – implicants number growth

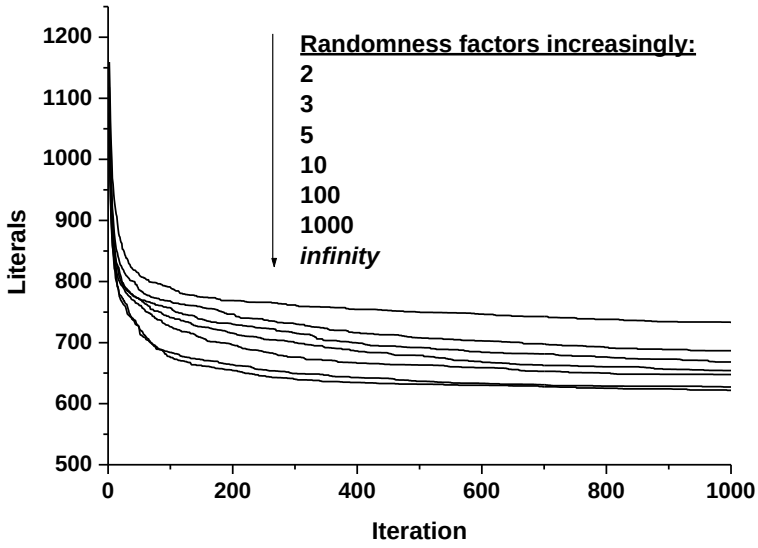


Figure 51. Derandomized FC-Min – solution quality

6.10 Asymptotic Completeness of the Algorithm

The completeness of the FC-Min search strategy is discussable. It is complete (up to the UCP solution algorithm completeness), if the **I** matrix contains minterms only. This is because of even though not any implicant can be generated in the Find Cover phase in general, the randomized Implicant Expansion phase is able to produce any prime implicant. Indeed, there is a non-zero probability that the Find Cover phase will leave any **I** matrix minterm untouched, thus any prime implicant can be obtained just by expanding the **I** matrix.

If terms of higher dimensions are present in the source, it may happen that some solutions will not be reached. An example of such a case is generating a *consensus* of two terms [6]. It requires a

reduction of one term, which is not performed in FC-Min. However, if an implicant reduction phase was included (see Subsection 5.2.3) in the overall process, FC-Min would be asymptotically complete too.

6.11 CONCLUSIONS

The last randomized iterative logic optimization algorithm presented was FC-Min, another SOP minimizer. FC-Min is especially efficient for functions with a large number of output variables; for single-output functions its behavior is completely chaotic.

Iteration is performed both on low and high level here. As for the high-level iteration, FC-Min can be run in the same way as BOOM; a valid solution is obtained in every iteration, whereas each iteration produces different solutions. The solution can be then combined by solving the covering problem. Since FC-Min is well scalable too, high-level iteration does not introduce any significant run-time overhead.

Low-level iteration and randomness are employed in the implicants generation phase. Randomness is in FC-Min used in a completely different way: the implicants generation phase run is *probabilistic*, the termination criterion of the Find Cover phase is driven randomly, with a given probability. This allows this phase generate any solution at all. In combination with low-level iteration, randomness offers a possibility of setting a trade-off between the solution quality and run-time.

As well as for BOOM, it was shown that FC-Min is able to outperform the state-of-the-art SOP minimization tool ESPRESSO – even though inferior results are usually obtained in the first iteration for complex problems, better solutions than ESPRESSO produces are obtained after several iterations of FC-Min.

In contrast to the algorithms mentioned in the previous sections, in the case of FC-Min the minimum required random number generator granularity cannot be analytically computed. In fact, any loss of randomness involves a loss of efficiency here. Zero randomness makes the algorithm end up in an infinite loop.

For more details see [89], [90], and [58].

Note that BOOM and FC-Min were later combined to form a universal SOP minimizer BOOM-II [91], [92]. However, description of BOOM-II is beyond the scope of this work, since its behavior, in terms of iteration and randomness, is retained from BOOM and FC-Min.

7 FINAL CONCLUSIONS AND DISCUSSION

Several different applications of randomness and iteration to logic synthesis and optimization were introduced and discussed. Logic optimization was understood as a general combinatorial optimization process; the notion of state space was introduced and discussed for particular algorithms.

Levels of iterations were defined. Iteration, as a part of single logic synthesis and optimization steps, was denoted as *low-level*. Conversely, the *high-level* iteration meant repeating the whole synthesis process, in order to improve the result quality in time.

Sources of "external randomness" were explored and its effects were documented. All examined logic synthesis tools (both commercial and academic) were found to be sensitive to ordering of variables in the source file submitted to the synthesis, commercial tools were even sensitive to the ordering of statements (code lines) in the source file. This means, results of different qualities (in terms of area) are obtained, if just functional equivalence non-disturbing perturbations of source code statements are applied. Possible reasons for such a behavior were discussed.

This fact was then further exploited in a method *systematically improving* the result quality. Randomness was non-violently introduced into the standard iterative synthesis process, so that standard synthesis tools can be used. This can be accomplished by randomly permuting variables before every iteration. A kind of *diversification* is introduced this way; locally optimum solutions are more likely escaped. As a result, the iterative power of the whole iterative synthesis process is increased, yielding systematically better solutions (both in area and time).

Even more diversification and randomness was introduced into the standard synthesis process, by synthesizing the circuit *by parts*. Randomly selected continuous parts of the circuit were extracted and submitted to synthesis separately. This enables producing a larger variety of different results, possibly due to obscuring misleading structures to the synthesis.

Even higher result quality was obtained, in terms of area. Experiments have shown that the process behaves rather consistently; the reached improvement was not a "lucky coincidence", i.e., the synthesis process is *robust*.

The amount of randomness (in terms of the random number generator granularity, not its quality) necessary to make the algorithm perform well was studied. It was shown that not much of it is actually required, however definitely the algorithm cannot be run in a deterministic way. Results obtained by deterministic runs were very inferior.

The minimum necessary random number generator granularity was analytically derived for this algorithm, as the number of nodes in the initial network.

Next, two randomized iterative sum-of-products (SOP) forms minimizers were introduced. In both cases, randomness is used to generate a variety of different implicants of the source function. These are accumulated in progress (high-level iteration) and the final solution is constructed using all of them, by solving the covering problem. The more implicants are found, the better solution can be obtained.

The first SOP minimizer, BOOM, does not require too much randomness to perform well. Actually, the minimum necessary random number generator granularity was analytically derived as well, as the number of variables in the minimized function.

In case of the second minimizer, FC-Min, randomness is additionally employed to drive its probabilistic run. Every single implicant is generated in a probabilistic way, finding a trade-off between its quality (ability to cover more on-set terms) and validity. If an invalid implicant is generated, the generation process is re-run. With no randomness employed, either very poor solutions are obtained, or the process never terminates. As the probability function is continuous, the amount of randomness has been found crucial here; the algorithm performs well only with lots of randomness employed.

All of the proposed algorithms have one thing in common: giving the *possibility* of obtaining better solutions, at expense of run-time. Therefore, generally it can be concluded that if there is enough time for synthesis, and/or results obtained by a conventional synthesis do not meet the designer's constraints (area, power consumption, delay, etc.), randomized iterative processes are a possible choice.

Unfortunately, iteration is not generally accepted by EDA vendors (i.e., producers of the EDA software). The major objection is the run-time increase. However, sometimes one can improve the result by orders of magnitude by applying several iterations only.

Conversely, hardware (ICs) designers (i.e., consumers of the EDA software) are not so strictly against iteration. Often they would be willing to spend more design time to reach better results, especially when low-power or low-area designs are required.

The main objection against randomness is the inability of reproducing the results; two runs of a randomized algorithm will return different results. However, in practice this may not be true as well. We are not speaking about true randomness; it is *pseudo-randomness*, actually. Thus, results can easily be reproduced by fixing the pseudo-random number generator seed. Also, the seed may be specified as a synthesis parameter (among the others, like optimization effort, etc.), enabling the designer produce possibly different solutions upon his wish.

Besides of acceptance, randomized iterative algorithms offer a possibility of obtaining upper bounds of complexity of circuits. This could be exploited in many areas, especially in research; lower bounds show the way and determine the target of synthesis.

To conclude, here are summarized the main features (pros & cons) of *randomized iterative algorithms*:

- they may be time-consuming,
- return unpredictable results (as the standard synthesis does too, actually),
- enable exploring a larger (possibly arbitrarily large) state space,
- increase the iterative power of algorithms (w.r.t. standard iterative processes),
- can be used to obtain *upper bounds* of the circuit complexity (or delay, power, etc.),
- enable discovering *different* solutions of one problem instance, possibly of the same quality. Then a secondary quality criterion can be applied over these results,
- offer a trade-off between quality and run-time.

REFERENCES

[1] D. Gajski and R. Kuhn, *Guest Editor's Introduction: New VLSI Tools,* in *Computer*, Vol. 16, No. 12, 1983, pp. 11-14.

[2] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*, Boston, MA, Kluwer Academic Publishers, 1996, 564 p.

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

[4] W.V. Quine, "The Problem of Simplifying Truth Functions," in *The American Mathematical Monthly*, vol. 59, No. 8, 1952, pp. 521-531.

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

[6] R.K. Brayton et al., *Logic Minimization Algorithms for VLSI Synthesis*, Boston, MA, Kluwer Academic Publishers, 1984, 192 p.

[7] R. L. Ashenhurst, "The Decomposition of Switching Functions," in Proc. of the International Symposium on the Theory of Switching, Part I, 29, 1957, pp. 74–116.

[8] H. A. Curtis, *A New Approach to the Design of Switching Circuits*, Van Nostrand, Princeton, N.J., 1962, p. 653.

[9] J. P. Roth and R. M. Karp, "Minimization over Boolean Graphs," in *IBM Journal of Research and Development*, Apr. 1962, pp. 227–238.

[10] S. B. Akers, "Binary Decision Diagrams," in *IEEE Transactions on Computers*, vol. C-27, No. 6, June 1978, pp. 509-516.

[11] R. E. Bryant, "Graph Based Algorithms for Boolean Function Manipulation," in *IEEE Transactions on Computers*, Vol. 35, No. 8, August 1986, pp. 677-691.

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

[13] O. Coudert, „Doing Two-Level Logic Minimization 100 Times Faster," in Proc. of the 6th ACM-SIAM Symposium on Discrete Algorithms, San Francisco, California, USA, 1995, pp. 112-121.

[14] K. Karplus, "Using If-Then-Else DAG's for Multi-Level Logic Minimization," Univ. California, Santa Cruz, UCSC-CRL-88-29, Nov. 1988, p. 21.

[15] C. Yang and M. Ciesielski, "BDS: A BDD-Based Logic Optimization System," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 21, No. 7, 2002, pp. 866-876.

[16] R. K. Brayton, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert R.Wang, "MIS: a Multiple-Level Logic Optimization System," in *IEEE Transactions on Computer-Aided Design*, Vol. 6, No. 6, Nov. 1987, pp. 1062–1081.

[17] 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, p. 52.

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

[19] Berkeley Logic Synthesis and Verification Group, "ABC: A System for Sequential Synthesis and Verification" [Online]. Available: http://www.eecs.berkeley.edu/alanmi/abc/.

[20] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial Strength Verification Tool," in Proc. of the 22nd International Conference on Computer Aided Verification, Edinburgh, UK, July 15-19, 2010, LNCS 6174 6174, Springer 2010, pp. 24-40.

[21] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification," in *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 21, No. 12, December 2002, pp. 1377-1394.

[22] P. Bjesse and A. Boralv, "DAG-Aware Circuit Compression For Formal Verification," in Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2004, pp. 42-49.

[23] A. Biere, "AIGER", [Online]. Available: http://fmv.jku.at/aiger/.

[24] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "DAG-Aware AIG Rewriting: a Fresh Look at Combinational Logic Synthesis," in Proc. of the 43rd Design Automation Conference (DAC), San Francisco, CA, USA, 2006, pp. 532-535.

[25] A. Mishchenko and R. K. Brayton, "Scalable Logic Synthesis Using a Simple Circuit Structure," in Proc. of the 15th International Workshop on Logic and Synthesis (IWLS), Vail, Colorado, USA, June 7 - 9, 2006, pp. 15-22.

[26] A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to Technology Mapping for LUT-based FPGAs," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 26, No. 2, February 2007, pp. 240-253.

[27] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and Sequential Mapping with Priority Cuts," in Proc. of International Conference on Computer-Aided Design (ICCAD), November 5-8, 2007, San Jose, CA, USA, pp. 354-361.

[28] A. Mishchenko, R. K. Brayton, and S. Chatterjee, "Boolean Factoring and Decomposition of Logic Networks," in Proc. of International Conference on Computer-Aided Design (ICCAD), November 2-5, 2008, San Jose, CA, USA, pp. 38-44.

[29] R. K. Brayton et al., "SAT-based Logic Optimization and Resynthesis," in Proc. of International Workshop on Logic Synthesis (IWLS), 2007, pp. 358-364.

[30] A. Saldanha, A. Wang, R.K. Brayton, and A. L. Sangiovanni-Vincentelli, "Multi-Level Logic Simplification using Don't Cares and Filters," in Proc. of the 26th Design Automation Conference (DAC), Las Vegas, Nevada, USA, 25-29 June 1989, pp. 277–282.

[31] R.K. Brayton and C. McMullen, "The Decomposition and Factorization of Boolean Expressions," in Proc. of the International Symposium on Circuits and Systems (ISCAS), May 1982, pp. 49–54.

[32] H. Savoj and R.K. Brayton, "The Use of Observability and External Don't Cares for the Simplification of Multi-Level Networks," In Proc. of the 27th Design Automation Conference (DAC), Orlando, Florida, USA, 1990, pp. 297–301.

[33] R. J. Francis, J. Rose, and K. Chung, "Chortle: A Technology Mapping Program for Lookup Table-Based Field Programmable Gate Arrays," in Proc. of the 27th Design Automation Conference (DAC), Orlando, Florida, USA, June 24-28, 1990, pp. 613-619.

[34] J. Cong and Y. Ding, "FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vo. 13, No. 1, January 1994, pp. 1-12.

[35] P. Pan and C.-C. Lin, "A New Retiming-Based Technology Mapping Algorithm for LUT-Based FPGAs," in Proc. of ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays (FPGA), February 22-24, 1998, Monterey, CA, USA, pp. 35-42.

[36] G. Ausiello et al., *Complexity and Approximation*, Springer, Nov. 1999, p. 524.

[37] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co. New York, USA, 1990, p. 338.

[38] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984, p. 382.

[39] S. Anily and A. Federgruen, "Simulated Annealing Methods with General Acceptance Probabilities," in *Journal of Applied Probability*, Vol. 24, No. 3, Sep. 1987, pp. 657-667.

[40] S. B. Gelfand and S. K. Mitter, "Analysis of simulated annealing for optimization," in Proc. of 24th IEEE Conference on Decision and Control, Dec. 1985, pp.779-786.

[41] B. Hajek, "Cooling Schedules for Optimal Annealing," in *Mathematics of Operations Research*, Vol. 13, No. 2, May 1988, pp. 311-329.

[42] J. A. Darringer et al., "Logic Synthesis through Local Transformations," in *IBM Journal of Research and Development*, vol. 25, no. 4, July 1981, pp. 272-280.

[43] J.M. Sanchez and J. Lanchares, "Multilevel Logic Synthesis Using Algorithms Based on Natural Processes," in Proc. of the 20th International Conference on Microelectronics, 12-14 Sep. 1995, pp. 823-828.

[44] A. Kuehlmann, P. Färm, and E. Dubrova, "Logic Optimization Using Rule-Based Randomized Search," in Proc. of Asia and South Pacific Design Automation Conference (ASP-DAC), Shanghai, China, 18-21 Jan. 2005, pp. 998-1001.

[45] K. Ohmori and T. Kasai, "Logic Synthesis Using a Genetic Algorithm," in Proc. of IEEE International Conference on Intelligent Processing Systems, Beijing, China, 1997, pp. 137-142.

[46] Z. Vašíček and L. Sekanina: "Formal Verification of Candidate Solutions for Post-Synthesis Evolutionary Optimization in Evolvable Hardware," in *Genetic Programming and Evolvable Machines*, Springer, Vol. 12, No. 3, March 2011, pp. 305-327.

[47] P. Fišer, J. Schmidt, Z. Vašíček, and L. Sekanina, "On Logic Synthesis of Conventionally Hard to Synthesize Circuits Using Genetic Programming," in Proc. of the 13th IEEE Symposium on Design and Diagnostics of Electronic Systems (DDECS), Vienna, Austria, 14.-16.4. 2010, pp. 346-351.

[48] B.W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," in *Bell Systems Technical Journal*, Vol. 49, 1970, pp. 291–307.

[49] P. Fišer and J. Schmidt, "A Difficult Example or a Badly Represented One?" in Proc. of 10th International Workshop on Boolean Problems (IWSBP), Freiberg, Germany, 19.-21.9.2012, pp. 115-122.

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

[51] D. Brand, "Hill Climbing with Reduced State Space," in Proc. of IEEE International Conference on Computer-Aided Design (ICCAD), 7-10 Nov. 1988, pp. 294-297.

[52] A.N. Kolmogorov, "Three Approaches to the Quantitative Definition of Information," in *Problems Information Transmission*, Vol. 1, No. 1, 1965, pp. 1–7.

[53] R. Smolensky, "Algebraic Methods in the Theory of Lower Bounds for Boolean Circuit Complexity," in Proc. of the 19th annual ACM symposium on Theory of computing (STOC), ACM, New York, NY, USA, 1987, pp. 77-82.

[54] A. Razborov, "Lower Bounds for the Size of Circuits of Bounded Depth with Basis {∧, ⊕}," in *Math. notes of the Academy of Sciences of the USSR*, No. 41, 1987, pp. 333-338.

[55] R. Beigel and J. Tarui, "On ACC," in Proc. of 32nd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Washington, DC, 1991, pp. 783-792.

[56] [Personal communication with EDA industry]

[57] [Personal communication with ASIC design industry]

[58] P. Fišer and J. Schmidt, "How Much Randomness Makes a Tool Randomized?," in Proc. of the 20th International Workshop on Logic and Synthesis (IWLS), San Diego, California, USA, 3.-5.6.2011, pp. 136-143.

[59] P. Fišer and J. Schmidt, "On Using Permutation of Variables to Improve the Iterative Power of Resynthesis," in Proc. of the 10th International Workshop on Boolean Problems (IWSBP), Freiberg, Germany, 19.-21.9.2012, pp. 107-114.

[60] A. Puggelli, T. Welp, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "Are Logic Synthesis Tools Robust?," in Proc. of the 48th ACM/EDAC/IEEE Design Automation Conference (DAC), 5-9 June 2011, pp. 633-638.

[61] B. Bollig and I. Wegener, "Improving the Variable Ordering of OBDDs is NP Complete," in *IEEE Transactions on Computers*, Vol. 45, No. 9, September 1996, pp. 993–1002.

[62] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," in Proc. of the International Conference on Computer-Aided Design (ICCAD), Santa Clara, CA, 1993, pp. 42-47.

[63] F. Somenzi, "CUDD: CU Decision Diagram Package Release 2.4.1," University of Colorado at Boulder, http://vlsi.colorado.edu/~fabio/CUDD [Online].

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

[65] K. McElvain, "IWLS93 Benchmark Set: Version 4.0," Mentor Graphics, May 15, 1993, p. 6.

[66] S. Yang, "Logic Synthesis and Optimization Benchmarks User Guide," Technical Report 1991-IWLS-UG-Saeyang, MCNC, Research Triangle Park, NC, January 1991, p. 45.

[67] A. Mishchenko, S. Chatterjee, R. Brayton, X. Wang, and T. Kam, "Technology Mapping with Boolean Matching, Supergates and Choices," ERL Technical Report, EECS Dept., UC Berkeley, March 2005, p. 7.

[68] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing Structural Bias in Technology Mapping," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 25, No. 12, December 2006, pp. 2894-2903.

[69] V. Bertacco, S. Plaza, "Disjoint Support Decompositions from BDDs through Symbolic Kernels," in Proc. of Asia and South Pacific Design Automation Conference (ASP-DAC), Shanghai, China, 2005, pp. 276-279.

[70] V. N. Kravets and P. Kudva, "Implicit Enumeration of Structural Changes in Circuit Optimization," in Proc. of the 41st Design Automation Conference (DAC), San Diego, California, USA, 7-11 July 2004, pp. 438-441.

[71] http://opencores.org

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

[73] P. Fišer and J. Schmidt, "Improving the Iterative Power of Resynthesis," in Proc. of 15th IEEE Symposium on Design and Diagnostics of Electronic Systems (DDECS), Tallinn, Estonia, 18.-20.4.2012, pp. 30-33.

[74] P. Raghavan, "Probabilistic Construction of Deterministic Algorithms: Approximating Packing Integer Programs", in *Journal of Computer and System Sciences*, Vol. 37, No. 2, 1988, pp. 130-143.

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

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

[77] M. Chatterjee and D.K. Pradhan, "A Novel Pattern Generator for Near-Perfect Fault Coverage," in Proc. of the 13th IEEE VLSI Test Symposium (VTS), 30 Apr.-3 May 1995, pp. 417-425.

[78] N.A. Touba, E.J. McCluskey "Altering a Pseudo-random Bit Sequence for Scan-Based BIST," in Proc. of International Test Conference (ITC), 1996, pp. 167-175.

[79] Y. Tang et al. "X-Masking During Logic BIST and Its Impact on Defect Coverage," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 14, Issue 2, February 2006, pp. 193-202.

[80] P. Fišer and J. Hlavička, "Column-Matching Based BIST Design Method," in Proc. of the 7th IEEE Europian Test Workshop (ETW), Corfu, Greece, 26.-29.5.2002, pp. 15-16.

[81] P. Fišer and H. Kubátová, "Pseudo-Random Pattern Generator Design for Column Matching BIST," *Microprocessors and Microsystems journal*, Dependability and Testing of Modern Digital Systems special issue, Elsevier, vol. 32, Issues 5-6, August 2008, pp. 340-350.

[82] V. Gherman et al. "Efficient Pattern Mapping for Deterministic Logic BIST," in Proc. of the International Test Conference (ITC), 26-28 Oct. 2004, pp. 48-56.

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

[84] O. Coudert, "On solving covering problems," in Proc. of the 33rd Design Automation Conference (DAC), 3-7 Jun, 1996, pp. 197-202.

[85] E. Goldberg, L. Carloni, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, "Negative Thinking by Incremental Problem Solving: Application to Unate Covering," in Proc. of International Conference on Computer-Aided Design (ICCAD), Washington DC, USA, 1997, pp. 91-98.

[86] L. P. Carloni, E. I. Goldberg, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Aura II: Combining Negative Thinking and Branch-and-Bound in Unate Covering Problems," in Proc. of International Conference on Very Large Scale Integration: Systems on a Chip, 1999, pp. 346-361.

[87] M. A. Harrison, *Introduction to Switching and Automata Theory*, McGraw-Hill, 1965, p. 499.

[88] A. Bernasconi, V. Ciriani, P. Fišer, and G. Trucco, "Weighted Don't Cares," in Proc. of the 10th International Workshop on Boolean Problems (IWSBP), Freiberg, Germany, 19.-21.9.2012, pp. 123-130.

[89] P. Fišer, J. Hlavička, and H. Kubátová, "FC-Min: A Fast Multi-Output Boolean Minimizer," in Proc. the 29th Euromicro Symposium on Digital Systems Design (DSD), Antalya, Turkey, 1.-6.9. 2003, pp. 451-454.

[90] P. Fišer and H. Kubátová, "Boolean Minimizer FC-Min: Coverage Finding Process," in Proc. of the 30th Euromicro Symposium on Digital Systems Design (DSD), Rennes, France, 31.8.-3.9. 2004, pp. 152-159.

[91] P. Fišer and H. Kubátová, "Two-Level Boolean Minimizer BOOM-II," in Proc. of the 6th International Workshop on Boolean Problems (IWSBP), Freiberg, Germany, 23.-24.9.2004, pp. 221-228.

[92] P. Fišer and H. Kubátová, "Flexible Two-Level Boolean Minimizer BOOM-II and Its Applications," in Proc. of the 9th Euromicro Conference on Digital Systems Design (DSD), Cavtat, Croatia, 30.8.-1.9.2006, pp. 369-376.

LIST OF ABBREVIATIONS AND ACRONYMS

ABC – A system for sequential synthesis and verification
AIG – And-Inverter Graph
ALU – Arithmetic and Logic Unit
ASIC – Application Specific Integrated Circuit
ATPG – Automatic Test Patterns Generator
BDD – Binary Decision Diagram
BIST – Built-In Self-Test
BOOM – Boolean Minimizer
CD-Search – Coverage-Directed Search
DC – Don't Care
EDA – Electronic Design Automation
ESPRESSO - Heuristic logic minimizer
FC-Min – Find-Cover based Boolean Minimizer
FPGA – Field Programmable Logic Array
GA – Genetic Algorithm
HDL – Hardware Description Language
IC – Integrated Circuit
LFSR – Linear Feedback Shift Register
LSS – An IBM system for production logic synthesis
LUT – Look-Up Table
MVSIS – Multivalued SIS
NP – Non-Deterministically Polynomial
RTL – Register Transfer Language
VHDL – VHSIC Hardware Description Language
VHSIC – Very-High-Speed Integrated Circuit
PI – Primary Input; Prime Implicant
PO – Primary Output
PLA – Programmable Logic Array
SA – Simulated Annealing
SIS – Sequential Interactive System
SOP – Sum-of-Products
UCP – Unate Covering Problem