Performance Problems in Logic Synthesis

by

Jan Schmidt

A habilitation thesis submitted to the Faculty of Information Technology, Czech Technical University in Prague

June 2015

Copyright © 2015 Jan Schmidt
Abstract

Logic synthesis is a procedure transforming circuit specification to a logically equivalent and supposedly optimal implementation at the gate level. For at least three decades, automated logic synthesis tools are used in industry.

Two serious problems with performance of logic synthesis tools were reported. The first one concerned parity predictors, i.e., a class of circuits with a parity tree at output. The second one was with artificial but realistic circuits. Those two cases were similar: the upper bound for circuit area was known, the resulting area was orders of magnitude larger than the upper bound, and the original structure of the circuit had been replaced by an artificial and possibly ineffective one.

The reports gained attention but no solutions appeared. The problems may have the implicit promise that the overall synthesis performance could greatly be improved; or, as the industry noted, they are possibly irrelevant. To solve this dilemma, investigation was performed.

To evaluate the practical relevance of such phenomenon, the evaluation procedure itself must be relevant. We found that the interface to logic synthesis and hence the requirements to logic synthesis are not well defined. Moreover, the authenticity of some popular benchmarks is at least questionable.

We proved first that the problems with parity predictors are no coincidence, that they persist even after changes in the circuit. They are not linked to possible symmetries in the circuit. The discarded structure, however, can be rediscovered by decomposition tools, which are capable for efficient XOR circuit synthesis.

In the other problematic situation, the missing structure is also the key, since the procedures tested work with gradual improvements of the actual input structure. When the optimization fails to achieve a good structure, the output depends strongly on the input. This holds for any circuit transformation that discards the original structure. We quantitatively modeled such behaviors for different transformations.

We conclude that any synthesis tools should not be evaluated outside their intended use, which implies that the intended use must have been specified. Artificially transformed circuits can discover the inner working of a procedure but are of little worth in practical evaluation. Circuits used for evaluation must be authentic, including their derivation from designer’s specification.
Preface

“The question is this: Monsieur the Principal thinks that my thesis ought to be dogmatic and didactic.”

“Your thesis! Are you then making a thesis?”

“Without doubt,” replied the Jesuit. “In the examination which precedes ordination, a thesis is always a requisite.”

[...]

“Now,” continued Aramis, [...] “now, as you have heard, d’Artagnan, Monsieur the Principal is desirous that my thesis should be dogmatic, while I, for my part, would rather it should be ideal. This is the reason why Monsieur the Principal has proposed to me the following subject, which has not yet been treated upon, and in which I perceive there is matter for magnificent elaboration – ‘UTRAQUE MANUS IN BENEDICENDO CLERICIS INFERIORIBUS NECESSARIA EST.’ ”

[...]

“I do justice to the beauties of this thesis; but at the same time I perceive it would be overwhelming for me. I had chosen this text – tell me, dear d’Artagnan, if it is not to your taste – ‘NON INUTILE EST DESIDERIUM IN OBLATIONE’; that is, ‘A little regret is not unsuitable in an offering to the Lord.’ ” “Stop there!” cried the Jesuit, “for that thesis touches closely upon heresy. [...] You approach that famous point of free will which is a mortal rock. You face the insinuations of the Pelagians and the semi-Pelagians.”

Alexandre Dumas
The Three Mosqueteers

This work is a kind of luxury. It is a luxury of having a problem and going wherever facts and experiments lead to in order to understand it and explain it. It is a luxury of working without a pressure to produce immediately tangible results, sheltered from many regulations and policies, which are sometimes as intricate and fierce as the politics in Aramis’ times.

In the sense of Aramis, the work is ideal, although I am afraid that the reader will find it much less than ideal in the everyday sense. It certainly is not intended to be dogmatic (in our sense); the assertions presented here should not exceed their experimental support. It will be a honour if the work succeeds in its didactic mission; for example, if it improves current evaluation practice.

A question, as controversial as the notion of free will in Aramis’ thesis, is whether this work is quantitative or qualitative. Qualitative research is a greater heresy to scientific method than was Protestantism to Monsieur the Principal. Yet understanding is a quality. However, the experiments presented here are quantitative by necessity. The work, therefore, tries to build a bridge from quantitative to qualitative domain. Let the reader walk across it in peace.
Acknowledgement

During the long journey to this work, I had had a friendly and helpful companion, who was always eager to discuss the true meaning of an observation, had deeper knowledge of synthesis algorithms than I, and who also provided the experimental material. I thank for all this Petr Fišer, my colleague at CTU in Prague.

Such a long journey by paths far from mainstream would be impossible without a tolerant and understanding environment. For this, I am grateful to CTU in Prague, and both the Faculties of Electrical Engineering and Information Technology.

A key part of the habitat proved to be the hospitality of David Kovařík and his winery, where our endless talks circled around qualitative meaning of quantitative findings, often resulting in daring theories and conjectures.

Seeing that the work is fairly non-standard, I was sometimes lost in doubts. At that times, I met several people who gave me the necessary encouragement. I thank Ilya Levin of Tel-Aviv University, who agreed that given a problem, one has to set out to find explanation by all means, even if the effort takes them out of the well-trodden paths. At the Boolean Problems workshop (IWSBP 2014), I found the courage to speak about the connection between qualitative and quantitative aspects of my work. I thank Christian Posthoff, University of The West Indies, for appreciating the talk; it was a surprise, since he is a mathematician. At the International Workshop on Logic and Synthesis, I was surprised again by the interest of best researchers in the field, although my work was in an early stage then.

The quantitative “side” of the work is massively quantitative indeed, in the sense that massive amounts of machine time were needed. To give the reader an idea, one experiment (Section 8.4) needed 357,500 runs of a non-trivial synthesis procedure. In earlier stages of the work, the Faculty of Electrical Engineering provided most of the resources. Lateron, computational resources were provided by the MetaCentrum under the program LM2010005 and the CERIT-SC under the program Centre CERIT Scientific Cloud, part of the Operational Program Research and Development for Innovations, Reg. no. CZ.1.05/3.2.00/08.0144.

Many people provided valuable help for the work. Jason Cong and Kirill Minkovich (then at University of California in Los Angeles) provided files for the $CD(G25)$ example, which we were not able to produce. Maciej Ciesielski, University of Massachusetts, gave us the source code for BDS, a decomposition tool that proved principal for some of the experiments. Alan Mishchenko, University of California in Berkeley, in reaction to our research, programmed the permute command in ABC, and translated many circuit examples so that they could be processed by academic tools. Last but not least, my colleagues, Martin Novotný and Jan Pospíšil, helped me to discover how commercial tools process arithmetic circuits.
# Contents

1 Introduction
   1.1 Motivation .................................................. 3
   1.2 Problem Statement .......................................... 3
   1.3 Contributions of the Thesis ................................ 3
   1.4 Structure of the Thesis .................................... 4

2 Digital design
   2.1 Decomposition .............................................. 7
   2.2 Synthetic and analytic steps ............................... 9
   2.3 Domains of description and the Y-diagram .............. 9
   2.4 The flow of digital design ................................ 10
   2.5 Summary .................................................... 12

3 Boolean functions and their representations
   3.1 Terminology ................................................. 14
      3.1.1 Boolean functions ...................................... 14
      3.1.2 Representation .......................................... 15
      3.1.3 Boolean formula ......................................... 15
      3.1.4 Boolean network ......................................... 16
      3.1.5 NPN classes .............................................. 18
   3.2 A representation of a Boolean function as a search state  18
      3.2.1 Flexibility of representation .......................... 18
      3.2.2 Measures ................................................ 18
      3.2.3 Distance ................................................. 19
   3.3 Summary .................................................... 20

4 Combinational circuits
   4.1 Practical circuits ........................................... 21
      4.1.1 What is a practical circuit? ............................ 21
   4.2 Description languages ..................................... 24
   4.3 Benchmark sets ............................................. 27
   4.4 Summary .................................................... 31

5 Logic synthesis and technology mapping
   5.1 Main tasks and principles .................................. 33
   5.2 Function representations ................................... 34
   5.3 Bias .......................................................... 35
   5.4 Logic synthesis from textual descriptions .............. 37
      5.4.1 Information in textual description .................. 37
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIG</td>
<td>And-Inverter Graph</td>
<td>17, 20, 59, 58, 59, 77</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
<td>33, 42</td>
</tr>
<tr>
<td>ATPG</td>
<td>Automatic Test Pattern Generation</td>
<td>25, 27, 100, 107</td>
</tr>
<tr>
<td>BDD</td>
<td>Binary Decision Diagram</td>
<td>18, 20, 51, 55, 98, 100, 104, 107</td>
</tr>
<tr>
<td>BDS</td>
<td>BDD-Based Logic Optimization System</td>
<td>89</td>
</tr>
<tr>
<td>BED</td>
<td>Boolean Expression Diagram</td>
<td>59</td>
</tr>
<tr>
<td>BIST</td>
<td>Built-in Self Test</td>
<td>47</td>
</tr>
<tr>
<td>CGP</td>
<td>Cartesian Genetic Programming</td>
<td>72, 91</td>
</tr>
<tr>
<td>DAG</td>
<td>directed acyclic graph</td>
<td>16, 34, 51</td>
</tr>
<tr>
<td>DC</td>
<td>Don’t Care</td>
<td>24, 25, 28, 42, 45</td>
</tr>
<tr>
<td>DFC</td>
<td>Decomposed Function Cardinality</td>
<td>23, 49, 50</td>
</tr>
<tr>
<td>DSOP</td>
<td>Disjoint Sum of Products</td>
<td>16, 100</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
<td>38, 40</td>
</tr>
<tr>
<td>EDA</td>
<td>Electronic Design Automation</td>
<td>2, 6, 12, 18, 24, 37, 60, 67, 70, 75, 76, 78, 80, 81, 84</td>
</tr>
<tr>
<td>ESL</td>
<td>Electronic System Level</td>
<td>11, 13</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
<td>2, 19, 33, 35, 38, 40, 42, 43, 49, 60, 61</td>
</tr>
<tr>
<td>FRAIG</td>
<td>Functionally Reduced And-Inverter Graph</td>
<td>55, 36, 59</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
<td>8, 10, 27, 37, 39, 45, 75, 79, 80</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
<td>10, 24, 29, 34, 37, 50, 42, 45</td>
</tr>
<tr>
<td>IP Core</td>
<td>Intellectual Property Core</td>
<td>38, 39</td>
</tr>
<tr>
<td>LEKO</td>
<td>Logic Example with Known Optimum</td>
<td>73, 79</td>
</tr>
</tbody>
</table>
LEKU Logic Example with Known Upper bound. 73 79
LUT Look-up Table.  12 19 25 33 43 49 59 61 82 83 89 100 102
OAC OpenAccess Coalition. 24
PLA Programmable Logic Array. 35 42 78
QBF Quantified Boolean Formula. 61
RBC Reduced Boolean Circuit.  59
ROBDD Reduced Ordered Binary Decision Diagram. 18 34 71 75
RSS Residual Sum of Squares. 102 103 107
RTL Register Transfer Level. 11 21 29 111 113
SAT Satisfiability Problem.  20
SOP Sum of Products. 16 34 35 42 45 82 89 94 97 98 100 102
Chapter 1

Introduction

Recent society increasingly depends on performance of electronic devices for communication and computing. More and more, these devices are digital. They have to deliver the required performance, but not at all cost. We do care how much batteries in a mobile device last, we do care how much computing performance can be put into a server room with limited heat removal facilities.

Integrated circuits are crucial in this effort. Ancient computers based on discrete components showed the limits of performance and reliability. Digital circuits with $10^9$ active elements are routinely manufactured. Although they contain regular structures (such as memories) and repeated blocks (such as processor cores), the complexity of parts to be designed is enormous.

Long ago, when transistor count just had risen over $10^4$, designers were forced to abandon their ingenious, hand-crafted designs, and reluctantly started to rely on Electronic Design Automation (EDA). In recent work flows, designers do not modify the design manually (perhaps with the exception of critical analog blocks). Instead, they guide the automated tools by setting constraints and optimization criteria. As a consequence, the resulting circuits are as good as the tools and their users are.

Full-scale integrated circuit design is limited to high-volume, high-performance devices, such as processors and mobile phone chipsets. The rest of digital devices, especially small-volume or experimental circuits, are built using programmable devices, mostly Field Programmable Gate Arrays (FPGAs). Performance losses due to programmability are compensated by advanced technology used to fabricate these devices, and by logistic advantages, such as drastically lower non-recurring cost and easy upgrade.

Design for FPGAs is (and must be) much simpler than the design of classical integrated circuits. As Steve Trimberger of Xilinx put it at the FPL’06 conference: “We struggle with deep-submicron design so that you don’t have to.” The tools are simpler, and to promote their devices, FPGA vendors provide them at low cost (which is frowned upon by the rest of EDA vendors). FPGA designers do not have the outstanding qualifications of integrated circuits designers, and therefore the degree of automation is even more important. FPGA tools permit “push-button synthesis”, as the ‘real’ designers call it, not without pejorative connotations. It means that the design synthesis is fully automatic, requires less expertise, and even so produces good quality results. In this case, however, the quality of tools determines the quality of result almost exclusively.

As automated design procedures matured, the initial opposition from designers ceased, and design tools gained trust. It is trusted that the resulting product is of reasonable quality, for example, that battery life could be improved at most by ten percent, with manual effort that no
1.1. MOTIVATION

one can afford. Fifty percent, of course, would raise concern, and even bigger figures would force
the supplier of the tools to investigate the causes and improve the tools. Unfortunately, there is
no way to estimate the true optimum solution in practice.

1.1 Motivation

Amidst this rather rosy picture, experiments were reported which demonstrated that a well-
developed and trusted part of the design tools, namely logic synthesis, did not always perform
as well as supposed, and that the quality margin was dramatic.

Cong and Minkovich (2007b) published a family of artificial circuits, for which they were
able to prove upper bounds on circuit area. They also proved that all known metrics of those
examples are similar to practical circuits. Synthesis results obtained from both academic and
commercial synthesis systems were far over that bound, sometimes by a factor of 400. So far, no
explanation of the performance problems has been provided, nor do we know whether they are
significant.

architecture of self-checking circuits, which is based on predicting and checking parity at the
outputs. It was assumed that a parity predictor will be smaller than the original circuit, since
the synthesis will be able to merge the circuit with the parity tree and simplify it. This indeed
happened with many examples, yet there were circuits, for which the predictor was much larger
then the original circuit plus a parity tree. Such a situation is very similar to the preceding one,
as we have a natural upper bound on size. Moreover, these are practical circuits, not artificial
ones.

Fišer (2012) and Fišer and Schmidt (2012b) studied methods to improve the robustness of
synthesis tools against examples similar to the above mentioned, at the cost of execution time.
Also, Fišer, Schmidt, Vašíček, et al. (2010) used computationally intensive but robust heuristics
to these problems. Using such enhanced tools, new upper bounds were found for known examples.
They demonstrated that the quality margin of commonly reported results is dramatic.

1.2 Problem Statement

The primary problem of this work is to understand the above performance problems, and to
explain them. We need to answer the following questions:

• Are the observed phenomena important for practice?
• Can we understand the causes of the problems?
• Can we overcome, improve or circumvent the causes of the problems?
• If the causes are improved, will it lead to substantial improvements of synthesis algorithms
  (as, e.g., is hoped for in Cong and Minkovich (2007b)?

1.3 Contributions of the Thesis

1. We proved that structural bias is the cause of the problems.

2. We modeled how a heuristic behaves under circuit transformations that occurred in the
   problematic situations.

3. We found experimental evidence that the model differs between transformation types.
4. We conclude that synthesis algorithms have been (not always consciously) adapted to practical input, and should not be used to process dissimilar inputs, including the problematic examples.

5. We also conclude that the classical (decomposition-based) approach to synthesis is able to overcome the structural bias, where its limited scalability allows.

1.4 Structure of the Thesis

The presented work aims to understand some undesirable phenomena observed in practical or nearly-practical situations. This is not a well-known well-defined problem; for explanations, we had to analyze multiple areas, and only the combination of all of them forms an integral view and brings a presentable result. In some of the areas, we had to stress facts that were known but rarely presented. Sometimes, only the source code of academic tools gave us the needed answers. Therefore, the logic structure is complicated.

The observed performance problems result from interactions between circuits and logic synthesis procedures in the framework of digital design. All those areas had to be studied.

Chapter 2 outlines the design process and the principal engineer’s tools to manage complex task – abstraction, decomposition, and hierarchy. Perhaps unlike other areas of engineering, synthetic and analytic design steps are distinguished more sharply. Also, distinguishing structure from behavior is a key concept, needed in all subsequent explanations.

Chapter 4 has two main goals. The first one is to discuss the question What are practical circuits, which was first asked by Shannon in 1949 (Shannon 1949b) and has not been satisfactorily answered yet. Another goal of the chapter concerns circuit metrics. As Chapter 6 discusses, any interpretation of an experiment needs metrics of input data, result data, and resources. We present commonly used metrics. Some of them measure Boolean functions, some measure representations of Boolean functions.

The chapter also deals with benchmarks in a manner that could be called benchmark archeology. It shows how the origin and history of a particular benchmark circuit can affect its suitability for a particular purpose. Also, what would contemporary software practitioners call benchmark ontology is analyzed. We would not like to create the impression that this work should have been titled “The fruits of poor benchmarking culture”, however, the facts presented here are crucial to judge relevance of findings presented later.

Chapter 5 determines the position of logic synthesis in contemporary design process. We document that the interface to logic synthesis changes with the development of technology and in synthesis itself, and that the requirements to logic synthesis are determined by its context.

Algorithms of logic synthesis are heuristic, and biased towards solutions of certain nature or properties. Most of sub par results obtained from the heuristics can be traced down to some bias of the algorithm used to produce the result, and so the sources of bias had to be analyzed.

There are three main approaches to logic synthesis: classical (decomposition-oriented), rule-based, and resynthesis-based. They come from different periods of time, yet they do not form a clear evolutionary sequence in the sense that newer procedures overshadow the older ones. More recent procedures are better adapted to contemporary requirements, but the older ones still have abilities that the newer ones lack. Later on, we document that these abilities are beneficial in the problematic situations we study.

Instead of Chapter 6, we would like to direct the reader to some well established text on experimental algorithmics; we found, to our surprise, none that would entirely cover the needed
1.4. STRUCTURE OF THE THESIS

Figure 1.1: Idea diagram of the thesis
areas. In this chapter, the main principles of trustworthy experimental evaluation of EDA procedures are established. The fact presented earlier—that the class of practical circuits cannot be reliably characterized—limits the methods we can use, since hardly any assumption—statistical or otherwise—on the circuits can be drawn. In such a situation, we must require that circuit examples should be of authentic origin. When measured by such criteria, current benchmark circuit sets are found insufficient.

Main analysis of two performance problems follow in Chapter 7 and Chapter 8. Each chapter exposes the problem, and then a series of experiments aimed to explain the reasons follow. For each experiment, we describe the experimental setting, results, and discuss its interpretation. The general conclusion is that structural bias is the culprit. In Chapter 8, we suggest models to characterize the reaction of a procedure to structural changes of the input. We conclude that the tools are not intended to handle radical changes in the structure, and fail to optimize beyond a certain limit.

The consequences of the findings for design, evaluation and future research are summarized in Chapter 9.
Chapter 2

Digital design

Digital design is an engineering process that starts with the initial specification of an artifact, which we call circuit and finally produces a description of an implementation of the circuit, conforming the specification.

The initial specification, obtained e.g. from a customer, is usually in plain language. Its requirements can be divided into two broad categories.

Constraints. The core constraint for a circuit is its function. Any quantitative measure of the final device can also be constrained: throughput, latency, power, size, cost, etc.

Optimization criteria. The quantitative measures can be also prescribed for optimization.

Thus, we want e.g. the cost of a device to be the lowest possible and not exceeding the negotiated value; or we want to have maximum throughput with given silicon area.

The final implementation is always machine-readable, as it is used either to program a field programmable device, or in manufacturing an integrated circuit. Mask description and device test are the two most important components of implementation data.

2.1 Decomposition

Decomposition is a term which is used in many meanings, often in a single text. Here, we aim to clarify artifact and task decomposition; the decomposition of Boolean functions will be covered later (Chapters 4.1 and 5.6.3).

A recent integrated circuit can contain over one billion of individual elements. Neither humans nor information technology is able to describe and handle designs of such size as a whole. As with other engineering tasks, the following, related, techniques are used to overcome:

Artifact decomposition and hierarchy. The entire circuit is described as a composition of parts. When done recursively, a tree structure, a hierarchy, results. Parts used repeatedly can be described only once. Such a description is called a module, an entity or a block type in various circumstances, and the use of it an instance of an entity or a block. We avoid the term cell altogether, as it has been heavily overloaded in the past, with meanings that were incompatible and confusing.

Task decomposition and hierarchy. Similarly to an artifact, the design process can (and must) be decomposed. Two ways of decomposition stand out:
Parallel task decomposition. It may happen that some parts of the design task can be worked on concurrently. From the project organization point of view, this is desirable, as larger teams can be employed and the time to market is shorter. At the end, the results of concurrent efforts must be composed into a single result. This is called integration in the design community. Parallel task decomposition may be the result of artifact decomposition, sometimes the artifact is decomposed because of possible task decomposition. Another reason may be that part of the work has already been done, as the part of the artifact is taken over from earlier project, or is brought in from a third party, or generated automatically.

Serial task decomposition. In this setting, the task is decomposed into subsequent parts or phases. The results of one phase serve as the specifications of a consequent phase.

Parallel and serial task decomposition have also their analogies in combinatorial optimization. Parallel decomposition is used in global methods. They decompose a problem’s instance into two (smaller) instances, solve them (recursively) and construct a solution of the original instance as a composition of them (Servit and Zamazal [1995]). Serial task decomposition can be found in local iterative heuristics, which find solutions by a large number of successive transformations (Pearl [1984], Chapter 2).

Abstraction. Abstraction hides details that are not necessary for a particular task and simplify both the description and the task. The initial specification of a circuit can be seen as the most abstract one; by adding design decision, we arrive at the most concrete description, the implementation. Serial task decomposition mentioned above is often accompanied by gradual decrease of abstraction level, as the prevailing design style is top-down.

Some levels important for the design process have been named:

Algorithmic level. At this level, the function of a circuit is described in terms of processes, variables, and operations on the values of variables.

Register transfer level. The circuit is described as a composition of a data part (datapath) and a control part. The data part moves data between abstract memory locations through stateless operational units. The control part is a Finite State Machine (FSM) which receives flags from the data part, and issues control signals to it.

Gate level. The circuit is a set of Boolean functions and memory elements (flip-flops). All signals are discrete.

Transistor level. The circuit is composed of active and passive elements, the signals are continuous.

Abstraction and decompositions do enable us to manage complex designs, but at the cost of quality. A parallel decomposition may e.g. omit a chance to share a common entity. More significantly, to decompose means also to design communication between the decomposed parts. In some cases, this is a trivial task, but in others, communication overheads may impair the entire system performance.

With serial task decomposition, keeping (near-) optimality of the results is even more difficult. Each step produces a result, which, when taken as a specification, should lead to good results in subsequent steps. For example, when designing a gate-level schematic of a circuit, one must take into account delays in the interconnection. These delays can be accurately estimated only
with the knowledge of the physical form of the interconnection, which is, of course, still unknown at the time. Therefore, great number of estimations, often heuristic in nature, is used during a serially decomposed workflow.

2.2 Synthetic and analytic steps

So far, only steps bringing in new design decisions and approaching the final implementation have been considered. Yet, after each major step of the design, the circuit should be evaluated. The reasons are:

- **Constraints check.** The correctness of the design, that is, the conformance to functional specification, is checked by verification using simulation or formal methods. Quantitative measures get evaluated or estimated, and compared to constraints.

- **Suitability check.** Measures that estimate the suitability of the result for subsequent phases are calculated. For example, a design of a processor may ask for interconnection, which is too complex for the given technology.

True analytic steps produce, among other data, human-readable reports for further decision. Simulation, for example, can produce gigabytes of simulated traces, but the crucial output is the short message from the monitor procedures integrated into the testbench that all traces conform specification.

Another form of analytic procedure, which is much more important for this study, uses optimization criterion to guide the synthetic steps internally. Algorithms for synthetic steps are mostly heuristic, as the related optimization problems are NP-hard. Such heuristics, as we will discuss later, are frequently iterative. Various techniques can be used to lead an iterative heuristic to constraints conformance and criteria optimization. This is the case of, e.g. timing-driven synthesis.

Stand-alone analytic procedures can spend – and need to spend – relatively large run time. For example, design teams can wait for a timing conformance check for at least several hours, as this step runs ideally only once during the design. Guiding iterative procedures within synthetic steps needs much faster evaluation. In local heuristics, evaluation time of the optimization criterion prevails to such an extent, that the complexity of algorithms is compared by the number of optimization criterion evaluations only (e.g. Han et al. 2015).

2.3 Domains of description and the Y-diagram

At higher levels of abstraction, we tend to speak more of what the circuit does, that is, about its function or behavior. At somewhat lower level, we understand the circuit as a collection of smaller blocks with abstract interconnection. At the lowest level, physical implementation (such as the transistor size) is the focus. The correspondence, however, is rather loose.

Gajski and Kuhn 1983 suggested that the notions of abstraction level and description domain are in fact orthogonal. They did so in a guest editorial in a special issue of IEEE Computer on new VLSI tools, in an effort to provide a firm terminological basis for contributions in that issue. Their contribution to digital circuit ontology became known as the Y diagram (Figure 2.1) and is a part of most of the textbooks (e.g. Michel, Lauther, and Duzy 1992a).

At the system level, the behavior of a digital system may be given by an algorithm. When the first decomposition is done on the system, we have the top-level architecture, that is, the
structure of the system. At that moment, the available physical implementation space can be divided between the decomposed units (floorplanning).

A design procedure may stay in a domain, while refining the description to a lower level of abstraction. For example, an FSM given by a transition graph can be later described using Boolean expressions. Even the level of abstraction may remain the same, in the case of optimization.

A procedure producing a structure which realizes a given behavior is called synthesis in the broader sense. The most common term for a procedure converting an abstract structure into physical design is technology mapping.

![Figure 2.1: Gajski-Kuhn Y diagram](image)

In earlier textbooks such as Michel, Lauther, and Duzy 1992, analysis was depicted going from structure to behavior. Although such an approach suggests itself (analysis as the opposite to synthesis), it does not fit to analytical steps in recent designs. In the past, analysis consisted almost entirely of simulation, and for this the diagrams were correct. Recently, many more properties need to be checked and/or estimated, and the notion of analytical steps being the reverse of synthetic steps is lost.

The distinction between the behavioral and structural domain is much weaker in practice. Hardware Description Languages (HDLs) commonly used for synthesis, namely VHDL IEEE 1076 and Verilog IEEE 1364-2005, originated as simulation languages. Their respective authors intended to support mixed descriptions – behavioral and structural. At the RT level, certain behavioral constructs (at least FSM specification) are supported by synthesis tools. The common term ‘behavioral VHDL’ only means that all behavioral constructs can be used without limitation.

### 2.4 The flow of digital design

With the above explained terms, we can construct a relatively simple top-level view of the overall design procedure (Figure 2.2).
Register Transfer Level (RTL) specification has been considered as the standard starting point of design for two decades (Nekoogar and Nekooga 2003). With Electronic System Level (ESL) gaining popularity, the procedure can also start with system-level specification. Design space exploration and design refinement in higher-level synthesis is frequently manual, guided by high-level modeling, simulation, and estimations. For arithmetic-intensive designs, behavioral synthesis is also an option, including automated mapping of operations to operational units (space) and steps (time).

At this point of design, the following is known:

- **Operational units** in terms of their number and functionality.
- **Memory locations** and their implementation (registers, register banks, memory).
- **Data transfers** between operational units and memory locations; their implementation (bus-based, multiplexer-based).

The task is to translate all this into a gate-level structural description, which the back-end procedure of physical design can take. This is the task of **logic synthesis**.

The boundary between synthesis and physical design is not very well defined, and is con-
stantly shifting. Early EDA systems produced a technology-independent structure first, then transformed it into a technology-dependent form. The technology-independent structure consisted of abstract components; the technology-dependent structure was built from components known implementation in the target technology environment.

For example, it is known how to implement a multiplexer in any integrated circuit technology, or in a programmable device based on Look-up Tables (LUTs). Therefore, it can appear in the technology-independent structure. A technology-dependent structure would contain only elements actually available. For the integrated circuit, the multiplexer could be built from smaller multiplexers existing in a vendor’s library. For a programmable device, the multiplexes would be combined with other logic and implemented by a LUT primitive, which would then occur in the structure.

In this two-step procedure, the second step was usually called technology mapping, and initially considered as a part of the physical design. The need of good-quality results caused more and more knowledge of the target technology to be utilized during logic synthesis, and to even iterate technology dependent and independent steps. Currently, we understand that logic synthesis produces technology-adapted structures.

The final synthetic steps deal with physical implementation. The available physical space (a chip, a programmable device) is considered, all components placed there and interconnection implemented (as conductors, by programmable interconnect). Although this step is difficult and resource-intensive, it is outside the scope of this study.

2.5 Summary

- Since a non-trivial design is too complex, it is decomposed; this may be
  - task decomposition, which decomposes the procedure, or
  - artifact decomposition, which identifies parts of the designed artifact for processing.

- A decomposition can be
  - parallel, where the decomposed subtasks or parts can be processed independently and the results integrated, or
  - serial, where the results of one subtask or decomposed parts of a design are the specification for the other subtask or parts.

- Decomposition can lead to a hierarchy, and each level of the hierarchy can use a different level of abstraction.

- At each level of abstraction, the description of a circuit can be
  - behavioral, stating what the circuit does, or
  - structural, stating how the components are connected, or
  - physical, describing the exact shapes and positions of components and interconnect.

- Tasks in design can be
  - synthetic, which make design decisions, or
  - analytic, which evaluate either input specifications or results of synthetic steps.

- Design procedure is commonly decomposed into
2.5. **SUMMARY**

- **higher-level synthesis** (ESL or behavioral), which starts with system-level specifications and produces a design composed of memories, operational units and transfer paths,
- **logic synthesis** and **technology mapping** transforms the above into structure of abstract gates or implementable components,
- **physical design** decides on exact shape and placement of components and interconnection.
Chapter 3

Boolean functions and their representations

First of all, we had to reconcile the terminology used in mathematics (e.g. Jukna 2012) with that of synthesis textbooks (Brayton et al. 1984; Hassoun and Sasao 2002; Hachtel and Somenzi 1996). Certain properties of Boolean functions representations appear important for the following discussion, and are briefly recapitulated.

3.1 Terminology

3.1.1 Boolean functions

- A function $f : B^n \to B$, where $n$ is a natural number and $B$ is the carrier set of a Boolean algebra $(B, \{+, \cdot, \neg\})$, is a Boolean function. When not indicated otherwise, $B = \{0, 1\}$.

- A Boolean function $f : B_f \subset B^n \to B$, whose domain $B_f$ is a proper subset of $B^n$, is called an incompletely specified Boolean function.

- For a given Boolean function $f : B_f \subset B^n \to B$, any vector $X = x_1, x_2 \ldots x_n \in B_f$ is called a state of $f$.

- For a given Boolean function $f : B_f \subset B^n \to B$, any state $X$ of $f$ such that $f(X) = 1$ resp. $f(X) = 0$ is called a 1-state resp. a 0-state of $f$.

- For a given Boolean function $f : B^n \to B$, the set of all 1-states resp. 0-states of $f$ is called the on-set resp. off-set of $f$.

- An incompletely specified Boolean function $f : B_f \subset B^n \to B$ is commonly extended to $B^n$ by assuming that for any $X \in B^n \setminus B_f$, $f(X) = \text{DC}$, where DC stands for ‘don’t care’ and forms the third member of the co-domain $D = 0, 1, \text{DC}$. Other symbols such as ‘\neg’, ‘X’, or ‘*’ are used for DC as well.

- For incompletely specified Boolean function $f : B_f \subset B^n \to B$, $B_f$ is termed the DC-set, and any its member a DC-state.

- For two Boolean functions $f, g$, we write $f \preceq g$, iff, for any $X \in B_n$, $f(X) = 1$ implies $g(x) = 1$, and $f < g$ iff $f \neq g$. In both cases, we say that $g$ covers $f$. 

14
3.1. TERMINOLOGY

- Let \( X = x_1, x_2, \ldots, x_n \) be a vector in \( B^n \). Let us construct a vector \( X' \) by picking only selected members of \( X \). Then, we call \( X' \) a selection of \( X \).

- By picking not selected members, we obtain complement of \( X' \).

- Let \( X_1, X_2 \) be two selections of a vector \( X \). Then we define union and intersection of \( X_1 \) and \( X_2 \) in the standard set-theoretical way.

- Let \( f_1, f_2, \ldots, f_m \) be an ordered \( m \)-tuple of Boolean functions \( f_i : B^n \rightarrow B \) (that is, with a single common domain). Then, we call the \( m \)-tuple a vector Boolean function or multi-output Boolean function \( f : B^n \rightarrow B^m \).

- Let \( P_1, P_2, \ldots, P_m, P_F \) be finite sets. Then a function \( F : P_1 \times P_2 \times \cdots \times P_m \rightarrow P_F \) is, in this context, called a multivalued function.

- If, in the above, \( P_F = \{0, 1\} \), then \( F \) is a multivalued Boolean function, and a Boolean algebra can be defined over \( P_1, P_2, \ldots, P_m \). (Brayton and Khatri 1999)

- If, in the above, \( P_F = \{0, 1, DC\} \), then \( F \) is called incompletely specified multivalued Boolean function.

- Let \( P_i \) be some of the sets \( P_1, P_2, \ldots, P_m, P_F \) mentioned above. An injective function \( E : P_i \rightarrow B^k \) for some integer \( k \) is then called an encoding of \( P_i \).

3.1.2 Representation

- A representation of a Boolean function is any structure from which the Boolean function can be reconstructed, that is, its value for any arguments can be determined.

- A representation is canonical, iff there is exactly one structure of a given kind for a given Boolean function (up to an isomorphism).

3.1.3 Boolean formula

- Let \((B, \{+,-,\})\) be a Boolean algebra. Let \( V \) be an arbitrarily large set of symbols representing variables in the domain \( B \). Then, we define Boolean formula as a string, that can be constructed using the following rules:
  - 0, 1, and members of \( V \) are Boolean formulas.
  - For Boolean formulas \( F \) and \( G \), \((F) + (G)\) and \((F)(G)\) are Boolean formulas.
  - For a Boolean formula \( F \), \((\neg(F))\) is a Boolean formula.

  Instead of \( \neg x \), \( x' \) and \( \bar{x} \) symbols are commonly used. Based on precedence of '\(-\)' over '+', and '+' over '·', parentheses may be omitted.

- A Boolean formula represents a Boolean function \( f \), iff, for every valuations of variables, there is at least one sequence of formula transformation by the axioms of Boolean algebra leading to the correct value of \( f \).

---

\(^1\)This twist tries to straighten up the situation where the arguments of a function are seen as a vector or as a set depending on context. The “set view” is common in discussions of algebraic decomposition and two-level minimization, where it is called the cube calculus.

\(^2\)Of course, mathematical functions have no outputs; this term became a convenient shorthand for a vector of Boolean functions describing a multi-output circuit.

\(^3\)Contrary to the common use of “multivalued function”, cf. Borges 1967.
• An occurrence of any \( v \in V \) or its negation \( \neg v \) is a \textit{literal}.

• A Boolean formula \( Q \) having the form of a product of literals is called a \textit{product term}.

• A Boolean formula \( Q \) having the form of a sum of literals is called a \textit{sum term}.

• A Boolean formula having the form of a sum of product terms is called a \textit{Sum of Products} (SOP).

• For a given Boolean function \( f \), we call an \textit{implicant} any Boolean function \( q \leq f \), that can be represented by Boolean formula \( Q \) in the form of a product term.

• Two implicants \( p \) and \( q \) of a Boolean function \( f \) are called \textit{disjoint} iff there is no implicant \( r \) such that \( r \leq p \) and \( r \leq q \).

• A SOP is called \textit{Disjoint Sum of Products (DSOP)} iff all its terms are pairwise disjoint.

• A Boolean formula where any negation occurs only with literals, is in \textit{factored form}.

• An incompletely specified Boolean function can be represented by giving at least two of the three sets of states: on-set, off-set, DC-set. This is often written in two-column tabular form, where the left-hand column identifies a product term, the right-hand column identifies the state set. Notice that the value of the represented function in a state may be implied by more than one product term.

• The above notation can be extended to represent a set of incompletely specified Boolean functions. There must be one right-hand column for each function. The value in the column classifies the left-hand product term as
  - a member of the 1-set,
  - a member of the 0-set,
  - a member of the DC-set, or
  - irrelevant.

This notation is the base of so called \textit{PLA form} (Brayton et al. [1984]).

3.1.4 \textbf{Boolean network}

• Let \( G = (V, E) \) be a \textit{directed acyclic graph (DAG)} where each node is valuated by a Boolean function, called \textit{node function}. Among those nodes, let \( p \) nodes be valuated by an identity function; those nodes shall have an input degree of zero. We call them \textit{input nodes}. For other nodes, let the input degree of a node shall equal to the number of arguments of its node function. Further, let \( q \) nodes have an output degree of zero. These are the \textit{output nodes}. Such a structure is called a \textit{Boolean network}.

• A Boolean network represents a multiple-output Boolean function \( f : B^p \rightarrow B^q \) by the following evaluation:
  - Assign the value of arguments to the corresponding input nodes.
  - For each node in topological order, evaluate its node function: use the value of \( j \)-th predecessor as the \( j \)-th argument to the node function, evaluate that function, and assign its value to the processed node. (Certain kinds of Boolean networks mark edges to denote value inversion.)
3.1. TERMINOLOGY

- The values assigned to the output nodes are the resulting values of the function.

- Two nodes in a Boolean network are distance- \( k \) if the shortest path between them spans \( k \) edges. (adapted from Mishchenko and Brayton 2006).

- The immediate predecessors of a node are called the fan-in of the node.

- The transitive closure of the fan-in relation of a node is called the transitive fan-in or input cone of the node.

- The subset of all nodes in a transitive fan-in which are distance- \( m \) or less from the original node, is the distance- \( m \) transitive fan-in of the original node.

- The immediate successors of a node are called the fan-out of the node.

- The transitive closure of the fan-out relation of a node is called the transitive fan-out or output cone of the node.

- The subset of all nodes in a transitive fan-out which are distance- \( m \) or less from the original node, is the distance- \( m \) transitive fan-out of the original node.

- A Boolean network, where each non-input node function is '. (AND), each non-input node has exactly 2 predecessors, and edges can have valuation denoting negation, is called an And-Inverter Graph (AIG) (Kuehlmann, Paruthi, et al. 2002).

- An AIG where no two nodes have identical pairs of predecessors is Structurally Hashed AIG (abbreviated as “strashed AIG”).

- An AIG where no two nodes represent identical subfunctions is Functionally Reduced AIG (FRAIG) (Mishchenko, Chatterjee, R. Jiang, et al. 2005).

- A Boolean network with one output node, where all node functions in non-input nodes are restricted to members of a certain set \( \Gamma \) (the base) of (usually symmetric) Boolean functions, is called a Boolean circuit.

- In circuit complexity theory, a Boolean formula is a Boolean circuit having tree structure. Boolean formulas defined above are therefore a special case with \( \Gamma = \{+,.,\neg\} \). When not indicated otherwise, the former definition of Boolean formula will be used.

- A cut of node \( n \) is a set of nodes of the network, such that each path from a network input to \( n \) passes through at least one node of the cut. The node \( n \) is the root of the cut, the others form the leaf set (Mishchenko and Brayton 2006).

- A cut is \( K \)-feasible or simply a \( K \)-cut if the number of nodes in the leaf set does not exceed \( K \) (Mishchenko and Brayton 2006). Remark. Nodes in a cut can be seen as a boundary of some region of the network.

- A multi-output cut is the union of the leaf set \( N_k \) of input nodes and the root set \( N_l \) of output nodes such that every path from network inputs to a node in \( N_l \) passes through a node in \( N_k \), and every path from a node in \( N_k \) to network outputs passes through a node in \( N_l \) (adapted from Martinello et al. 2010; Mishchenko and Brayton 2006).

- A multi-output cut is \( KL \)-feasible, iff \( |N_k| \leq K \) and \( |N_l| \geq L \) (adapted from Martinello et al. 2010). Remark. Thus, \( K \)-cuts are a special cases of \( KL \)-cuts for \( L = 1 \).
3.1.5 NPN classes

- Two Boolean functions are **P-equivalent**, iff they can be made equal by permuting their inputs.

- Two Boolean functions are **NPN-equivalent**, iff they can be made equal by
  - negating some of their arguments,
  - permuting some of their arguments,
  - negating one of the function’s value.

3.2 A representation of a Boolean function as a search state

Representations of Boolean functions are used for different purposes; we query the structures to determine whether two functions are identical, whether they belong to a single NPN class, and so on. We require those queries to be fast and reliable; canonical representations have advantages. Reduced Ordered Binary Decision Diagrams (ROBDDs) by R. E. Bryant (1986) are one prominent example.

Most of the optimization procedures discussed in this study are local search procedures. The object of optimization is a combinational circuit, therefore candidate solutions of the optimization are some representations of Boolean functions. Such a use is rarely discussed, yet it is the core of most EDA procedures.

3.2.1 Flexibility of representation

The transition from behavioral to structural domain (cf. Sections 5.1 and 5.6) is often aided by a structured representation of the implemented Boolean functions. The structure is gradually constructed and optimized, and then it is declared to be the structure of the circuit. In this process, the structural flexibility of the representation becomes important. Canonical representations offer no flexibility, and cannot serve as a basis for state space construction. They are, however, crucial in performing operations on subfunctions.

As synthesis transforms behavior to structure, the underlying representation must model both. Boolean networks do have structural (graph) and behavioral (node function) component. A node function is only a (single output) Boolean function. To model a general hardware structure with multi-output blocks, a separate structure model is usually employed, although multi-valued node functions could be used too. With separate structural information, node functions can be limited to Boolean formulas or even their normal forms; also, Binary Decision Diagrams (BDDs) can be chosen. In the case of general node functions, information can be shifted between behavioral and structural domain. Restricted classes of Boolean networks (AIGs, functionally reduced AIGs) do not offer this ability.

3.2.2 Measures

A circuit optimization procedure aims at some desirable properties of the future implementation, such as its silicon area or speed. These properties, however, depend on future design decisions during physical design, and can be only approximated at the level of logic synthesis and mapping. The representation should offer metrics that can serve for this purpose.

The number of literals in a Boolean network has been used to approximate the number of transistors implementing a given network, and hence to estimate its area. This approximation
has many sources of imprecision. Gates may be replicated or enlarged to drive high-fan-out signals, buffers and inverters inserted to speed up signal propagation. Yet this metric is still used in research, as it is simple and at least can serve for comparison.

The area in FPGAs is usually estimated by modeling the implementation of the Boolean network. A $K$-input LUT is represented either by a $K$-feasible cut, or by a node that has at most $K$ predecessors. This way, we assume that the target FPGA device is homogeneous, which does not hold since the construction of the early FPGA families. Look-up tables are combined with fixed logic such as carry chains, and the look-up tables are configurable to some extent in recent devices. All those features change the circuit implementation dramatically.

Similar remarks hold for timing, which is commonly estimated by the longest path through the network. In some technologies, logic elements have different delays, which moreover depends on their fan-out. Interconnect delay becomes more and more significant in recent technologies. Recent heterogeneous FPGAs have also heterogeneous interconnect delays – dedicated signal paths, such as carry chains, are significantly faster than the programmable signal paths.

### 3.2.3 Distance

Local optimization is based on state space traversal, which naturally leads to the idea of distance in the state space. Can this idea be formalized, that is, is there a metric? As the first step, we might define an arbitrary metrics on the set of states. As the next step, we might ask whether the operations of the algorithms (that is, the transitions in the state space) define a non-trivial metrics.

**Metric sets of states**

Here we limit ourselves to the simpler case where the states are representations of the same Boolean function, in other words, where the circuit is correct at any step of optimization.

A straightforward approach to this problem is to define an edit metrics on the set of states. For such a metric to exist, a set of transformations must be defined such that any state can be transformed into any other state using these transformations, and for each transformation, its inverse must also exist. The distance between two states is then defined as the minimum number of transformations between the two given states.

In the case of Boolean formulas (over the Boolean algebra $(\mathbb{B}, \{+, \_, \neg\})$), such a set of transformations is trivial. The axioms of Boolean algebra have the form of equalities. When we take those equalities as rewriting rules in both directions, we obtain a set of transformation conforming the requirements. Therefore, the set of equivalent Boolean formulas is metric.

Iwama and Hino (1994) extend this approach. The set of formulas over $(\mathbb{B}, \{\uparrow\})$ is considered. A formal mechanism to introduce function symbols is defined, and thus a Boolean network can be described. The completeness of the transformations is proved. An extension to other operators is mentioned, unfortunately in a report that is inaccessible to us.

Farm, Dubrova, and Kuehlmann (2005) proves the completeness of a set of transformations over AIGs, another special but important case of Boolean networks.

From these partial results, it seems that the set of equivalent general Boolean networks is indeed metric. By the metric subset property, subsets such as structurally hashed or functionally reduced AIGs are also metric. Therefore, it makes sense to speak about distance in sets of states independently of the discussed algorithm.
CHAPTER 3. BOOLEAN FUNCTIONS AND THEIR REPRESENTATIONS

Metric state spaces
Operations actually used by a heuristic can be very different from those defining an edit metric on the set of states. Mishchenko, Chatterjee, and Brayton (2006) replace a cut in an AIG by a predefined optimum graph; such an action can be quite “long” from the point of view of an edit metric on AIGs.

Operations of an algorithm can define an edit metric on their state space, providing that for each operation, its inverse also exists. Such a property is called a symmetric move structure. It is often assumed or required in advanced heuristics (Salamon, Sibani, and Frost 2002, Chapter 15). The requirement is sometimes relaxed in the sense that between any two states, forward and backward traversals shall take approximately the same number of operations. The existence of inverse operations is then an extreme form of the move structure symmetry.

Symmetric state spaces require a more advanced search control, as the state space itself does not prevent the algorithm to cycle. Simple, greedy algorithms cannot have symmetric state spaces, which is more usual with advanced heuristics such as simulated annealing (Kirkpatrick, Jr, and Vecchi 1983).

Using the algorithm’s operations to define an edit metric on the state space promises more relevant distance measurement. Not all algorithms, however, provide operations with the necessary properties.

Practical measurement
Experimental evaluation would benefit from the ability to actually measure the distance between two concrete representations. All edit metrics are difficult in this respect as they are defined as minima, that is, to find the distance is a combinatorial optimization problem. With representations of Boolean function, it is moreover a kind of proof of the equivalence, which is co-NP hard (Zhou, Singhal, and Aziz 1998). In practice, it is solved using BDDs or a solver for the Satisfiability Problem (SAT). Both methods have exponential upper bounds.

3.3 Summary
- We use standard textbook terminology for Boolean functions, plus terms common in the logic synthesis area for functions having multiple outputs in the Boolean domain or inputs/outputs in other domains.
- Representations of Boolean functions are either networks or decision diagrams. Limitations on what a node function in a Boolean network may comprise forms a hierarchy of restricted Boolean networks.
- NPN classes are an important notion for synthesis, as the members of a class can be transformed into each other by procedure which is trivial from the synthesis point of view.
- A synthesis procedure based on local search has a circuit representation as state. Restrictions on that representation restrict naturally the search space, and cause bias.
- Edit metrics can be introduced on some circuit representations, allowing us to measure the distance traveled by a local search procedure. The practicality and usefulness of such measurement is still an open question.
Chapter 4

Combinational circuits

This study deals with the interaction between circuits and algorithms. For experimental algorithmics, a model characterizing possible input would be needed. We summarize the answers to the question “what is a practical circuit?”. This question seems to have disappeared from discussion some time ago; yet having non-representative experimental material renders all experimental evaluation unreliable. We collected material from different sources, both famous and less-famous, to document our inability to answer the question reliably and to stress the need for authentic examples.

The manner in which the examples are described appears more important than as has been seen in the past. We document, on a brief domain analysis, that the languages commonly used are semantically disparate, and that almost any “conversion” between them loses authenticity and adds noise. This, in fact, did happen; we show on an overview of popular benchmark sets, that authenticity has been often neglected.

All these findings mean that authenticity is important. To see the degree to which common benchmark sets are authentic, we prepared an almost archaeological section on benchmarks origin and evolution. The result is not much enjoyable; numerous benchmarks shall be used with caution, as they are far from being authentic.

4.1 Practical circuits

4.1.1 What is a practical circuit?

This question dates back to Shannon [1949a]. By earlier theorem by Shannon, “most” circuits of $n$ inputs computing any function, the size of the circuit is asymptotically larger than $2^n/n$. Therefore, Shannon asks, why most real circuits are much smaller, and belong to a small group outside the “most”. The answer by Shannon [1949a, page 89] is that the human designer thinks of functions that can be explained and that follow some design patterns. The impact on implementation is apparent:

In general, the more we can decompose a synthesis problem into a combination of simple problems, the simpler the final circuits.
CHAPTER 4. COMBINATIONAL CIRCUITS

Practical circuits are decomposable

Shannon formalizes the decomposition of a Boolean function \( f(x_1 \ldots x_n) \) as the construction of two Boolean functions \( g \) and \( h \) such that

\[
f = h(g(x_1 \ldots x_s), x_{s+1} \ldots x_n), 1 < s < n - 1
\]  

(4.1)

and calls this functional relation **functional separability**. In recent terminology, this is **disjoint decomposition**, and includes **Ashenhurst decomposition** (Ashenhurst [1959]; Perkowski and Grygiel [1995]). Shannon nevertheless proves that the fraction of functionally separable functions approaches zero as \( n \) approaches \( \infty \).

That suggests, that practical functions are also in the small minority of **decomposable functions**. Besides marking a function as “possibly practical”, decomposition is an important step in implementation (Section 5.6.3).

Practical circuits have group invariants

Still with practical implementations in mind, Shannon identifies another helpful case:

**We will now consider a second type of functional relation which often occurs in practice and aids in economical realization. This type of relation may be called group invariance [...].**

After Shannon [1949a], let us define the following relations on Boolean functions of \( n \) variables

- \( N_{00...0} = I \) to be the identity relation;
- \( N_{10...0} \) to be the relation where one of the functions has its first argument inverted;
- \( \ldots \)
- \( N_{00...1} \) to be the relation where one of the functions has its \( n \)-th argument inverted;
- \( P_{12...n} = I \) to be also the identity relation;
- \( P_{21...n} \) to be the relation where one of the functions has its first and second argument swapped;
- \( \ldots \)
- \( P_{n2...1} \) to be the relation where one of the functions has its first and \( n \)-th argument swapped.

Then the relations \( N_i \) form an (Abelian) group under composition; \( P_j \) also form a group. For any \( N_i \) and \( P_j \), \( P_k \) can be found such that, for any \( f \)

\[
N_i P_j f = P_k N_i.
\]  

(4.2)

A Boolean function \( f \) is said to have **group invariance**, iff there are \( N_i \) and \( P_j \) such that

\[
N_i P_j f = f
\]  

(4.3)

Shannon comments on the importance of group invariance (Shannon [1949a] page 96)

*Of course, group invariance can often be recognized directly from circuit requirements in a design problem.*

Notice that **symmetry** (McCluskey [1956]) and **partial symmetry** are special cases of group invariance.
4.1. PRACTICAL CIRCUITS

Practical circuits are easy to describe

Sagalovich (1961) extends this approach by analyzing some 1000 circuits from communications and computing. He states that the circuits fall into the following categories:

- Functions having group invariance (Shannon 1949a).
- Decomposable functions (Povarov 1954).
- Non-repeated forms, where each argument is used exactly once; e.g. iredundant Maitra cascades (Maitra 1962).
- “Special” functions, having the decomposition
  \[ f(x_1 \ldots x_n) = \hat{x}_i f_1(x_1 \ldots x_{i-1}, x_{i+1} \ldots x_n) + f_2(x_1 \ldots x_i - 1, x_{i+1} \ldots x_n) \]  
  where \( \hat{x} \) is either \( x \) or \( \bar{x} \). This is a special case of decomposition as defined by Shannon.

Sagalovich then finds the minimum information which distinguishes a function within its class. For example, if we know that a function is (fully) symmetric, we only need to know the counts of arguments that must have the value 1 so that the function also gives 1, to fully determine the function.

From a recent point of view, an identification of a function’s class together with the minimum distinguishing information forms a kind of description complexity. It suggests that practical functions have a concise description (of some sort).

Design follows patterns

Ross, Noviskey, and Taylor (1991) takes Shannon’s conjecture about the role of decomposition much further. The authors designed a general decomposition scheme, implemented it in a software system, and tried to decompose not only Boolean functions from electronic design, but also from other fields such as pattern recognition. They came to the conclusion that human design indeed follows patterns, and that these patterns can be recognized and utilized.

Decomposition minimizes information flow

In (4.1), we assumed that \( g \) is a Boolean function. That can be generalized to a multi-valued function, requiring \( r \) bits to encode. Then the \( g \) function actually compresses \( s \) bits on input into \( r \) bits on output. Assuming \( r < s \), this is a gain for the implementation and possible guidance for the decomposition process. The amount of information processed by the decomposed function can be formalized by Decomposed Function Cardinality (DFC) (Perkowski and Grygiel 1995).

DFC of a block with \( n \) inputs and \( m \) outputs is \( m \cdot 2^n \). DFC of a function decomposed into multiple blocks is the sum of DFC of the particular blocks. In the case of non-disjoint decomposition, DFC is computed for each valuation of shared inputs.

Similar approach was taken by Jóźwiak (1997). Other metrics were used to characterize designed circuits. Rent’s exponent (Landman and Russo 1971) takes into account physical implementations. In the theoretically possible range from 0 (a chain of blocks) to 1 (a clique), most practical circuits has the exponent around \( p = 0.55 \); some “difficult” circuits approach \( p = 0.75 \), while randomly generated circuits tend to have \( p = 1 \).

What is considered a practical circuit?

To complement these older efforts with a recent approach, let us quote from W. Yang, L. Wang, and Mishchenko (2012):
For larger values of \( N \), only the functions occurring frequently in the benchmarks should be considered. We call such functions **practical** and give frequency statistics in the experimental results section.

We see that the attitude has reversed through the years: we cannot say what a practical circuit is, and so we depend on a choice of real examples. In the past, benchmarks were just challenges, and if they did not follow practice closely, no one cared much.

We are going to demonstrate in Chapter 5, however, that logic synthesis constantly adapts to environment and requirements. To work with artificial examples (without saying they are artificial) means leading the process to misadaptation.

The authenticity of examples plays a greater role than thought previously. In the following sections, we are going to see whether the authenticity can be preserved, and whether it was so with existing examples in the past.

### 4.2 Description languages

Since early nineties, design is HDL-oriented. Texts in description languages replaced schematic diagrams, since they were more concise and easier to write. Because of pressure from users (and despite vendors’ effort), these languages were standardized. Much earlier, academia also set up description languages for the exchange of examples and designs (e.g. Berkeley Logic Synthesis and Verification Group 1993; D. Bryant 1988). Later, the Silicon Integration Initiative (Silicon Integration Initiative 2005) founded the [OpenAccess Coalition (OAC) (OpenAccess Coalition 2005)](OpenAccess Coalition 2005), that, at that time, included several universities with an interest in [EDA]. The coalition defined not a language, but a programming interface simply called [OpenAccess](OpenAccess). The netlist descriptions of the IWLS’2005 benchmarks (Albrecht 2005) were released in this form only. During the years, university members left OAC and the link was broken.

We will limit ourselves to the following languages, which are actually used for industrial design and for academic purposes. We will discuss only properties that influence the description of combinational circuits synthesis. In some cases, such features form only a small subset of the entire language. The languages (or their discussed subsets) are all understood as “netlist descriptions”, with the exception of PLA, which is included because it was originally used to describe a number of benchmarks. The discussed properties are summarized in Table 4.1.

Hassoun and Sasao (2002, Chapter 3, p. 84) point out that current HDLs, or, more precisely, their synthesis subsets are not suitable for describing incompletely specified functions. Braud, Bergamaschi, and Stok (1995) distinguish two uses of Don’t Cares (DCs): comparison and assignment. Using don’t cares in an comparison permits the designer to write more compact conditions and may be considered a mere synthactic improvement. Assigning a don’t care to a signal, on the other hand, is the basic step in describing an incompletely specified function. Synthesizing such an assignment needs provisions in the language, and support from synthesis tools. Both may or may not be provided, depending on the language and tool.

**Verilog** ([IEEE 1364-2005](IEEE 1364-2005)) is a general hardware description language, originally intended for simulation but with a defined RTL synthesis subset. It describes both structure and behavior in a hierarchy. For combinational behavior, it officially lacks the notion of DC in assignment ([IEEE 1364-2005](IEEE 1364-2005)). Some tools (Synplicity, Inc. 2002) circumvent this deficiency by using the ‘bx’ value.

**VHDL** ([IEEE 1076.6](IEEE 1076.6)) is a language playing the same roles as Verilog, but having stronger modularity and parametrization. Signal values used for synthesis do comprise the ‘-’ symbol for ‘don’t care’. Many tools (Synplicity, Inc. 2002; Altera, Inc. 2015; UG 687) support
4.2. DESCRIPTION LANGUAGES

incompletely specified functions. From the point of view of this study, the two languages do not otherwise significantly differ.

**EDIF 2 0 0 NETLIST** (ANSI/EIA-548-1988) is one of the first netlist interchange standards that the industry accepted. It separates domains of description (here called *views*) in a manner similar to Gajski and Kuhn (1983). Only the NETLIST view is commonly used. It is a purely structural description; the view can describe any network of general (multi-output) blocks in hierarchical manner. The behavior of blocks that do not have a structural description (primitive blocks) is supposed to come from an external library. In benchmarking practice, a suitable library must be provided for each tool, or the behavior of gates must be guessed from their names. In the case of LUTs, vendor-specific extensions and conventions are used. For a truly creative way to express the behavioral concept of DC in structural EDIF, refer to Figure 4.3.

**PLA** or so called Espresso format (S. Yang 1991b, Section 4.3.1) is a language which describes a multiple-output incompletely specified Boolean function as a set of implicants and associated function values (Section 3.1.3).

**BENCH** (D. Bryant 1988) is an academic format constructed for Automatic Test Pattern Generation (ATPG) examples. In correspondence with the level of early test generation tasks, it describes a general network of pre-defined single-output logic gates and simple abstract registers.

**BLIF** (S. Yang 1991b, Section 4.2.2) basically describes a Boolean network (the base representation of the SIS system (Sentovitch and al. 1992)). It has provisions for register descriptions and – where the tools support it – hierarchical descriptions.

In the standard – and advisable – scenario, the examples come from industry and are subsequently used to test various academic tools. Given the evolution of industrial methods, the languages are going to change. To gather a collection of benchmarks, the problem of “common ground” appears.

Saying that all the languages (or, more precisely, their subsets of interest) are seen as “netlist descriptions”, implies that they share common terms, and the word “conversion” comes naturally to mind. Yet, as seen above, the terms the languages are built on are surprisingly different, their precise semantics differ greatly, and instead of information-preserving *conversion*, we have *transformation* that must take design decisions. Table 4.2 documents that such a case is common.

In production (design) systems, this could be tolerated; the transformations make “good” decisions, which do not harm the quality too much. In benchmarking, the situation is different. New algorithms shall be tested on tasks they will perform in practice, mostly taking human-designed circuits as inputs. Such an input filtered through numerous transformations is no longer genuine.

Another difficulty will be documented and analyzed in Section 5.4. It is not evident which information will be put to general (random) logic synthesis, and what will be processed by specialized procedures. The choice differs between systems and changes in time. Arithmetic circuits, which can cause much trouble in logic synthesis, are an example. Some recent design systems assume that their logic synthesis will be exposed to adders construction, some do not.

For those reasons, circuit examples must be taken cautiously, especially in cases where they cause difficulty. Their authenticity should be beyond doubt, or they would be considered irrelevant. As we will show in the next section, such a loss of authenticity has happened already.
### Table 4.1: Principal characteristics of languages describing combinational circuits

<table>
<thead>
<tr>
<th>Environment</th>
<th>Format</th>
<th>Structure</th>
<th>(node) behavior</th>
<th>incompletely spec.</th>
<th>hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>industry</td>
<td>Verilog</td>
<td>general network</td>
<td>expressions</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>VHDL</td>
<td>general network</td>
<td>expressions</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>EDIF 2 0 0</td>
<td>general network</td>
<td>no (ext. library)</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>academic</td>
<td>PLA</td>
<td>no</td>
<td>multi-output PLA</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>BENCH</td>
<td>Boolean network</td>
<td>predefined gates</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>BLIF</td>
<td>Boolean network</td>
<td>single-output PLA</td>
<td>yes (ext. DC)</td>
<td>yes (via models)</td>
</tr>
</tbody>
</table>

### Table 4.2: Non-trivial transformations (beyond syntactic conversions) between languages describing combinational circuits

<table>
<thead>
<tr>
<th>From ↓ To →</th>
<th>VHDL/Verilog after elaboration</th>
<th>EDIF 2 0 0</th>
<th>PLA</th>
<th>BENCH</th>
<th>BLIF</th>
</tr>
</thead>
<tbody>
<tr>
<td>VHDL/Verilog after elaboration</td>
<td>–</td>
<td>expressions to gates synthesis</td>
<td>collapsing; minimization</td>
<td>expressions to gates synthesis</td>
<td>expressions: node func. or structure</td>
</tr>
<tr>
<td>EDIF 2 0 0</td>
<td>node func. guess</td>
<td>–</td>
<td>node func. guess; collapsing; minimization</td>
<td>node func. guess; expressions to gates synthesis</td>
<td>node func. guess; output separation</td>
</tr>
<tr>
<td>PLA</td>
<td>output separation; minimization; DC?</td>
<td>two-level to gates; no DC</td>
<td>–</td>
<td>two-level to gates; no DC</td>
<td>output separation</td>
</tr>
<tr>
<td>BENCH</td>
<td>none needed</td>
<td>gate func. to gate name</td>
<td>collapsing; minimization</td>
<td>–</td>
<td>none needed</td>
</tr>
<tr>
<td>BLIF</td>
<td>node func. expression; external DC integration</td>
<td>node func. to gate synthesis; no external DC</td>
<td>collapsing; external DC integration</td>
<td>node func. guess; expressions to gates synthesis; no external DC</td>
<td>–</td>
</tr>
</tbody>
</table>
4.3 Benchmark sets

The first common sets of circuit examples appeared in mid-eighties. Two independent sets appeared: one set of multi-level circuits for ATPG and a set of two-level circuits to test logic minimizers. The second one, known as “Berkeley PLA set”, includes also contributions from University of Leuven. The problem is, that a part of those circuits are random. Rudell (1986) detected that from the logic minimization point of view, those examples have properties different from ordinary circuits (Section 6.5). Still, they were considered useful in minimizers evaluation. Harder to accept is the fact that the circuits were reused in later sets without any warning. They are available even in multi-level versions, without any documentation of the transforming process (which must have worked with and consume don’t cares in the original descriptions).

Other sets were targeted to sequential synthesis, mainly FSMs. After state encoding, transition functions are obtained, and it is tempting to use them as examples of combinational circuits. Yet their properties depend on the FSM synthesis procedure, and most notably, on the encoding used. A transition function designed for a minimum encoding (the 80’s standard) differs much from a function designed for one-hot encoding (common encoding for larger FSMs in recent designs).

In early nineties, EDIF appeared and the academia took an effort to adapt. As EDIF has no provisions to describe complex nodes of the existing Boolean networks, multi-level examples in the LGSynth’91 set (S. Yang 1991b) were resynthesized for the LGSynth’93 set (McElvain 1993) using a very impractical target library of gates. The synthesis tool was indicated as autologic in the files (probably unrelated to the automotive electronics vendor of the same name). Circuit names remained unchanged as the circuits were functionally equivalent.

![Figure 4.1: A comparison of circuits in the LGSynth91 and LGSynth93 sets. Dotted lines mark the 100:1, 10:1, and 1:1 size ratio.](image)

Figure 4.1 shows the consequences of the manipulation. For majority of circuits, their size measured in literals was not much affected, with the exception of the four circuits highlighted in the graph. They were enlarged to the extent that they no longer represent the same benchmark,
and we doubt if their structure is still representative for practical circuits. We were unable to replicate the effect of the manipulation even by old tools of the period. Practically all recent tools are unable to repair the harm in these cases, but usually are able to synthesize acceptable results from the original LGSynth91 circuit description.

We also studied the impact of the above mentioned manipulation to synthesis. Circuits from both sets were synthesized using the ABC system (Berkeley Logic Synthesis and Verification Group 2000) with the commands sequence `dch if lutpack` and are compared in Figure 4.2.

The descriptions of circuits `s27` and `s298` contained DCs in the form of a library cell named DC (Figure 4.3). These circuits could not have been synthesized. The problematic circuits `alu4` and `cordic` remained problematic, although the synthesis improved the ratio to LGSynth91. Surprisingly, new problems with circuits `frg1`, `too_large`, and `z4ml` appeared. It means that size alone is not sufficient to characterize the circuits. Furthermore, it forces us to mark the entire set as unreliable, because the altered structure of other circuits in the set can be difficult for

Figure 4.2: A comparison of circuits in the LGSynth91 and LGSynth93 sets after synthesis; `s27` and `s298` missing. Dotted lines mark the 100:1, 10:1, and 1:1 size ratio.

```plaintext
(cell DC
  (cellType GENERIC)
  (view INTERFACE
    (viewType NETLIST)
    (interface
      (port out (direction output))
    )
  )
)
```

Figure 4.3: An EDIF construct to express DC, `s27.edif.Z` in LGSynth93

`alu4` and `cordic` remained problematic, although the synthesis improved the ratio to LGSynth91. Surprisingly, new problems with circuits `frg1`, `too_large`, and `z4ml` appeared. It means that size alone is not sufficient to characterize the circuits. Furthermore, it forces us to mark the entire set as unreliable, because the altered structure of other circuits in the set can be difficult for
other tools, and would not test the tools for anything that can be present in practical circuits.

Table 4.3: A comparison of selected circuit as described in the LGSynth91 and LGSynth93 sets. All numbers are for LGSynth93 relative to LGSynth91.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Size, in literals</th>
<th>Size after synthesis, in LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>alu4</td>
<td>5.18</td>
<td>4.38</td>
</tr>
<tr>
<td>cordic</td>
<td>25.6</td>
<td>17.6</td>
</tr>
<tr>
<td>frg1</td>
<td>1.22</td>
<td>3.98</td>
</tr>
<tr>
<td>s27</td>
<td>5.83</td>
<td>–</td>
</tr>
<tr>
<td>s298</td>
<td>29.6</td>
<td>–</td>
</tr>
<tr>
<td>too_large</td>
<td>1.07</td>
<td>11.7</td>
</tr>
<tr>
<td>z4ml</td>
<td>1.29</td>
<td>4.63</td>
</tr>
</tbody>
</table>

It could seem that we give an unnecessary emphasis on mere five circuits in a large set, and that averaging over that set would diminish their influence. First of all, this study is about unpleasant surprises, and here a suspicion arises that some of the surprises are artificial and hence not really relevant. This applies not only to the benchmark set in discussion.

Second, we observed that some tools behave strangely on these examples. Performance histograms of tools working with *cordic* (here reproduced in Figure 6.9) started our research on performance modeling (Schmidt, Blažek, and Fišer 2014). *alu4* was one of the challenging examples selected for our study of iterative randomized synthesis procedures (Fišer, Schmidt, and Balcařek 2014a; Fišer 2012; Fišer and Schmidt 2014). It is not an encouraging thought that the work has little applicability to practice.

Around 2005, attempts were apparently undertaken to draw a thick line behind confused sets (Figure 4.4). The examples are from identified sources, and their purpose is known. The tools used to derive netlists from the original RTL descriptions are identified, although the actual choice of gates for logic synthesis is not documented, and the tools to process the OpenAccess format are not publicly available.

Mishchenko, Chatterjee, and Brayton (2006) comment on using the IWLS’2005 set:

*It should be noted that the original IWLS benchmarks were optimized by an industrial tool prior to distribution. [...] When starting with unoptimized networks, the difference between rewriting and traditional synthesis should be greater.*

They refer to the gate-level description in the OpenAccess format (Section 4.2). Because data in this format can no longer be used in academia, other tools are used for HDL elaboration. In the case of *QUIP* (Altera Corp. 2014), we have experimentally discovered that adders are translated to Boolean expressions and get mixed with other logic. Therefore, preserving authenticity during HDL elaboration is mostly shifted to the user.
Figure 4.4: The evolution of common benchmark sets
4.4 Summary

- Practical circuits cannot be characterized in a concise way. Characteristics such as decomposability, short description, group invariants, and design patterns do apply, yet they are insufficient to classify a circuit or to generate artificial examples.

- Languages describing any examples are semantically disparate, and in almost all cases an attempt to describe an example in another language needs design decisions, which loses authenticity.

- Existing benchmark sets are not fully authentic. Unreliable examples include:
  - Synthetic examples form University of Leuven, such as \textit{ex1010}.
  - FSM transition functions with unknown encoding and synthesis, such as \textit{s1488}, \textit{s1494}.
  - Multi-level example processed by inadequate synthesis, the multi-level section of LGSynth’93.
  - Newer gate-level benchmarks, if their derivation from the original RTL description is not reproducible.
Chapter 5

Logic synthesis and technology mapping

In this chapter, we analyze logic synthesis and technology mapping to establish several facts crucial for the performance problems discussed later. First of all, we observe that the distinction between logic synthesis and technology mapping is not sharp, and that the two share many features, procedures, and implementation problems.

The performance of synthesis procedure is principally limited by their heuristic nature. We show that the representations of Boolean functions play a role in the limitations, and that the limitations manifest themselves as a bias towards certain type of solution. Since bias appears to be closely related to the discussed problems, we describe several types of bias distinguished by their causes.

The primary goal of this chapter is to establish requirements to realistic experimental evaluation of synthesis procedures. Such an evaluation should mimic the real use of the procedure, and therefore should require what the real use requires, nothing more, but also nothing less. If some kind of input cannot get to a procedure in a real system, then to benchmark the procedure by such an input has a limited significance. We will show that procedures working in parallel with logic synthesis and technology mapping in real systems are numerous and widely used in practice.

Another goal is to analyze properties of various synthesis procedures with respect to performance. We are interested in what gives them the almost complementary strengths that manifested in our experiments. There are three approaches to logic synthesis, known as the classic one, the rule-based one, and the resynthesis-based one. A discussion of them shows that the differences between them result in different and complementary biases.

Most of the discussed procedures are based on iterative local search. In such a procedure, convergence control is crucial for performance. The authors of the procedures focused mainly on representations and techniques; little has been published on the top-level control of the respective tools. We analyze it to show its potential role in the performance problems.

For a general and detailed description of earlier procedures, we refer the reader to two classical textbooks, namely Hachtel and Somenzi (1996) and Hassoun and Sasao (2002). Not long after these books were written (circa 2006), what Ciesielski (2014) calls “the ABC revolution” came. Textbooks written after these years can hardly be found; the overview of these approaches is based on numerous research papers on this topic. We are interested in the iterative nature of these procedures; we extend the analysis by studying the high-level architecture of ABC commands used in our experiments from the ABC source code (Berkeley Logic Synthesis and Verification Group.
5.1 Main tasks and principles

Logic synthesis and technology mapping originated as clearly defined and separated procedures in Figure 5.1. The behavioral description of a circuit (e.g., by a Boolean expression) is converted into an optimum structure of abstract, technology-independent operators, such as AND, OR. The conversion is often based on a structured representation of a Boolean function (Section 3.2). After refinement and optimization, the structure of the description is declared to be the resulting circuit structure (cf. Section 5.6).

This structure is then implemented by blocks that can be directly realized in hardware. For Application Specific Integrated Circuit (ASIC) designs, they are the blocks which the technology providers offer in their library of cells. For FPGA implementations, they are the blocks actually existing on the FPGA chips, such as LUTs, carry structures, multiplexers, and fixed logic. As a consequence, the result of logic synthesis could be mapped to different target platforms without the need to repeat the synthesis procedure.

This beautifully clean and simple procedure, however, cannot work in practice. A general-purpose synthesis tool can hardly compete with specialized tools that can use higher-level information and produce much better results in their areas. In some cases, these tools can produce even physical representation of the circuit, bypassing the technology mapping stage. On the other hand, mapping tools can be incorporated into an iterative optimization procedure, as they transform the circuit in ways much different from logic synthesis.

Few academic systems have an architecture reflecting those facts, and few sources discuss this in detail. Apart various hints in reports produced by synthesis tools, the best source is Altera...
CHAPTER 5. LOGIC SYNTHESIS AND TECHNOLOGY MAPPING

5.2 Function representations in synthesis and mapping procedures

The procedures discussed here use information in all three domains (Figure 5.3). When we put aside complications caused by core generators in Figure 5.3, we get the following types of procedures by their domain traversals.

**Behavioral to structural.** The crux of (random) logic synthesis, in the original sense of a technology-independent procedure.

**Structural to physical.** The crux of technology mapping, in the original sense without any resynthesis.

**Refinement in any domain.** Deriving a representation at a lower level of abstraction in the same domain, possibly with some design decisions.

**Optimization in any domain.** Deriving a better representation at the same level of abstraction in the same domain, possibly with some design decisions.

A design system can have separate data structures for each domain. An example is the BDS decomposition system (C. Yang and Ciesielski 2002), which keeps behavioral information in (multiple) ROBDDs and the resulting structure in a DAG. Hachtel and Somenzi (1996, page 413) calls a similar approach “separated view”, although they do not distinguish synthesis from mapping and consider Boolean networks only.

In logic synthesis, the behavior is described by Boolean functions. A Boolean function is represented by some formalism (Section 3.2). That, in turn, is the subject of textual description in a HDL (Section 4.2). One Boolean function can have multiple structurally different circuits implementing it, and each circuit can have different descriptions in a given language. The differences in the latter case can be subtle, e.g. the order of variables, but even so they do have significant impact on synthesis procedures.

The flexibility problem has been introduced in Section 3.2. Many procedures represent Boolean functions by a representation which

- is structured (is able to express structure), and
- is not (entirely) canonical.

Then, both logic synthesis and mapping can be performed on this single representation (cf. Sections 3.2 and 5.6). This technique is called the “merged view” in Hachtel and Somenzi (1996, page 413). The input (behavioral) description is transformed, changing the structure of the description. At the final step, the structure of the representation is proclaimed to be the resulting structure of the circuit. For example, a Boolean network can be transformed until no node has a fan-in greater than a given constant $K$, and the result is then proclaimed to be a $K$-LUT implementation of the circuit. Examples of similar methods are summarized in Table 5.1.

An advantage of this approach, which is important for recent procedures, is the ability of resynthesis (Figure 5.2). During iterated synthesis and mapping, only the interpretation changes. For example, a Boolean network is understood once as a behavioral description and once as a structure.

On the other hand, unified representations also have drawbacks. In most cases, some post-processing must follow. For example, PLA generators have to search for identical terms in a SOP description of a multi-output function. For more complicated target environments, such as
Table 5.1: Examples of unified representations in synthesis and mapping

<table>
<thead>
<tr>
<th>Representation</th>
<th>Logic synthesis</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOP</td>
<td>AND-OR, evtl. NAND-</td>
<td>Programmable Logic Array</td>
</tr>
<tr>
<td>Boolean formula</td>
<td>NAND implementation</td>
<td>(PLA) structure</td>
</tr>
<tr>
<td>Boolean network</td>
<td>tree structure</td>
<td>gate implementation</td>
</tr>
<tr>
<td>general BDD</td>
<td>gate structure</td>
<td>gate/LUT implementation</td>
</tr>
<tr>
<td></td>
<td>MUX structure (rare)</td>
<td></td>
</tr>
</tbody>
</table>

5.3 Bias in synthesis and mapping algorithms

Exact algorithms produce always optimum results. When applied to circuits, we always get the circuit which is the best by the optimization criterion used, regardless of the initial circuit and the form of its description.

Heuristic algorithms, however, do not offer complete optimization and hence complete screening of the output from the input – in an extreme case, they fall under the popular excuse for poor performance: “garbage in, garbage out”. This is the case where structurally different (but functionally equivalent) circuits at the input produce different circuits at the output.

Files describing the same, functionally and structurally identical, circuit can vary in details such as the order of declarations, or the use of equivalent control constructs. Most of synthesis and mapping tools are sensitive to this variance (Puggelli et al. 2011; Shum and Anderson 2012; Fišer and Schmidt 2011; Fišer, Schmidt, and Balčárek 2014b). In this case, different descriptions of a circuit at input produce different circuits at output.

Randomized algorithms are yet another case: the same description of the same circuit at input produces different circuits at output.

Collectively, these influences are called bias, because the algorithm is biased towards some class or style of solutions. The main concern is that all solutions of reasonable quality fall into a class which the algorithm may be unable or unlikely to produce.

Perkowski and Grygid (1995) analyzed sources of bias in decomposition algorithms (Section 5.6.3). We augmented them with other sources, and generalized. The following list is not guaranteed to be complete, yet it covers most of the practical cases.

Representation bias From the combinatorial optimization point of view, the underlying representation influences the state space of local search algorithms, common in design tools (Section 3.2). It may also directly project into circuit structures the algorithm is able to produce (Section 5.2). A canonical representation offers no flexibility and allows no optimization; on the other end of the spectrum, representations such as general Boolean network can describe any structure. A representation ‘in between’, not entirely canonical but having internal restrictions, such as Functionally Reduced And-Inverter Graphs (FRAIGs) (Mishchenko, Chatterjee, R. Jiang, et al. 2005), affects a search in two ways. First, it makes the state space smaller, and the search works faster or more thoroughly. It is, however, a question whether the optimum solution (or any acceptable suboptimum solution) can be represented.

For example, we may ask if the optimum solution is represented by a FRAIG? It certainly depends on what we mean by “optimum”. We might want a description where the consecu-
tive technology mapping easily finds its target blocks, but FRAIGs force us to descriptions having a special AIG representation.

In other words, we ask whether the internal limitations of the representations are ‘good’ for the solution. Even if it is the case (as mostly it is), the state space may lack intermediate states, which make it easier for the search algorithm to reach an acceptable solution.

Heuristic operator bias. When a local heuristic is used, the reachability of states representing (sub)optimum circuits is important. State spaces with an asymmetric reachability relation are used to simplify the control of a heuristic. The optimality of results then strongly depends on the initial state of the search, although this case must be distinguished from state space pruning techniques, such as the Unate Recursive Paradigm (Brayton et al. 1984). Also global heuristics (where the solutions of simpler sub-instances are combined to produce a solution of the original instance) also may limit the choice of sub-instances. For example, recursive decomposition of a Boolean function may be limited to AND/OR decomposition only, or to disjunctive decomposition only (Section 5.6.3).

Distance bias. Every iterative heuristic is limited in the number of steps taken. The desired sub-optimum solutions may not lie within that limited number of steps, or may be unlikely reached by the number steps. Although the state space is not necessarily metric, the idea of “too long a distance” is often intuitive.

Requirement bias. An algorithm may also be limited by external requirements, such as testability (Steinbach and Stockert 1994). Often, we do not know what a (sub)optimum solution under these constraints is. Moreover, the algorithm may sacrifice more quality than necessary to conform such requirements.

Implementation bias. Multiple entities (such as Boolean network nodes, implicants, etc.) are processed in one iteration pass of a heuristic. The outcome of an entity modification taken in one such step can influence the following steps, as in the REDUCE procedure of ESPRESSO (Section 5.6.2). In a sequential algorithm, some order must be chosen. When there is no problem-oriented heuristic ordering, lexicographical ordering is commonly used. That ordering can follow, e.g., the order of declarations in the input file, or can be induced by the properties of the hashing function used to store node identifiers, and hence, by the identifiers themselves (Shum and Anderson 2012). Different but equivalent descriptions of a circuit at input then produce different circuits at output. To cure the cause, an unambiguous ordering heuristic is needed, which rarely can be found. To cure the consequences, randomization for fairness can be used, which replaces the (arbitrary) lexicographical ordering by random ordering. Such a cure, however, only converts implementation bias to random bias. For an exhaustive study of implementation bias, random bias and their utilization for better results, see Físer and Schmidt (2012c) and Físer (2012).

Random bias. Randomized algorithms (of the Monte Carlo type, e.g. based on simulated annealing) produce different circuits at output from the same description of the same circuit at input. This is the price for their ability to improve quality just by repetition, and other useful properties. In practical use, the variance is unwelcome, as the synthesis steps should be reproducible. For this reason, the randomized nature of practical algorithms is masked, e.g. by setting a fixed seed. For a designer, it is not a comfortable thought that much better result could have been obtained, only if the seed was different.

The biases listed above are, to a degree, inevitable. They can be seen as variance coming from different sources – the algorithm, the pseudorandom number generator, some obscure details of
ordering in the input file. From the algorithmic point of view, it is important how much the original variance is attenuated at the output. Intuitively, the better the heuristic, the greater the attenuation – an exact heuristic has no variance (measurable by the optimization criterion) at output.

5.4 Logic synthesis from textual descriptions

What is discussed in academia under the short term logic synthesis is, in fact, a rather limited area, which should be called random combinational logic synthesis. The terms custom logic synthesis or glue logic synthesis are also used, to distinguish the discipline from synthesis of regular structures with special properties. The synthesis of such structures is discussed less frequently, as it is mostly the matter of commercial EDA systems.

5.4.1 Information in textual description

HDLs are commonly used to describe circuits in large designs, since schematic representations are too cumbersome. Languages that can describe circuits for synthesis, such as VHDL [IEEE 1076.6] and Verilog [IEEE 1364-2005] are languages intended for large projects, concurrent engineering work, and reuse. In their standard form, they are not object-oriented. They are, however, all modular and support overloaded operators. In a typical setting, the first processing phase is elaboration, which integrates all modules, libraries and resolves overloading. At this point, no potentially useful information is lost. If the designer wishes the modular structure to be preserved, parallel decomposition takes place (Section 2.1).

As we mentioned in Chapter 2, these languages originated as simulation languages, with the support for mixed behavioral and structural description. Because this proved convenient for designers, this feature is supported by synthesis tools as far as is practical.

Simulation languages do not need any notion of state machines, unlike languages intended for synthesis from the beginning (ABEL 2003). It is therefore the task of a HDL synthesizer to recognize (or rather discover), which constructs actually describe a state machine. Then a procedure specialized for FSM synthesis is called. This is perhaps the most behaviorally oriented part of the input code.

Registers and memories are usually easy to discover in the description. The semantics of the RT-level notion of a register corresponds well to the function of a D flip-flop, and hence the implementation is straightforward. Not all registers are explicit, though. Sometimes only the complete set of assignments to a signal can tell us that the signal must have a state. In many cases, however, this is not what the designer wanted.

Memory is either represented as an array in the code, or instantiated explicitly as a library module at the request of the designer. In the first case, the memory is easy to discover. In both cases, a specialized algorithm is then used to implement the memory. Such a special module is just another example of a generator.

Hardware description languages are, as a rule, statically typed. The type carries useful information about the operations that can be performed with the value of a signal or variable, and therefore about the processing units required. What is a mere set of logic signals at the gate level, can represent a signed number at the RT level, and hence a generator for signed multipliers is called to implement the multiplication that occurred in the code. Such a situation must be recognized before the code is translated into Boolean functions, which are untyped.

A common pattern using type information in a HDL description is to mark specially a library function. For example, addition in VHDL code is elaborated to a call to a library function (called
plus in older packages), which is then marked using `pragma map_to_operator` (Synopsys and others) or `attribute foreign` (Xilinx). This way, synthesis tools are alerted of the special treatment, while simulators use the behavioral code of `plus`.

### 5.4.2 Parametric blocks and generators

Generators were much discussed during seventies and eighties (Allen 1986; Vai 2000) and helped to improve designers’ productivity (Michel, Lauther, and Duzy 1992b). At first, generators were intended to produce an optimized layout (physical description) for a class of circuits with given functionality, such as ripple-carry adders. When reuse and third-party intellectual property grew important, generators were used as representations of the reused or commercialized knowledge (Xilinx, Inc. 2011; Xilinx, Inc. 2012). Any reused or transferred module is called an Intellectual Property Core (IP Core), and the generators became “core generators.” For a given functionality, the user sets parameters such as the data width and other options. The generator returns any description of the circuit which is required by the design process, similarity to data characterizing a library element. Such a description may comprise

- structural information:
  - a netlist of abstract gates, or
  - a netlist of primitives in the target technology;
- physical information:
  - relative placement of blocks in the circuit, and/or
  - a complete physical representation in the target technology;
- behavioral information:
  - a simulation model, and
  - circuit timing, input load and sensitivity to output load.

Thus, the generators can be seen either as procedural representations of knowledge (e.g. how to build a proper adder), or heavily parametrized descriptions.

Circuits without any physical information are called “soft cores”, have the greatest flexibility, but the least degree of performance guarantee and the most pessimistic timing estimates. When only placement is provided, we speak about a “firm core”. A “hard core” has a complete physical representation, and therefore the best performance guarantees and timing estimates, at the cost of flexibility.

Soft cores may be technology-independent, described by abstract gates only, and therefore requiring subsequent technology mapping. This is often practical in ASIC design, where different technologies implement functionally similar gates, and the procedure of technology mapping is relatively simple. Netlists of technology-dependent primitives are less flexible, but can utilize special features of the target, such as carry chains or Digital Signal Processing (DSP) blocks in an FPGA, where they are commonly used.

In logic design, generators are used for adders, multipliers, and barrel shifters mainly (Bečvár 2014). To protect investment against rapidly changing technologies, ASIC core generators produce generic gates. FPGA generators produce descriptions targeted at a specific FPGA family; firm cores are common.

Memory generators for the ASIC environment also exist. They are tied to a particular technology, and even then the accuracy of generated timing models can be less than adequate (Bečvár 2014).
5.4.3 Practical design flow

Only the rest of the HDL code, namely combinational circuits that have no specialized generator, are actually passed to random logic synthesis. Besides the input code, such circuits commonly result from FSM synthesis, after (technology-dependent) state encoding (rarely optimized).

The resulting situation, as far as can be collected from the sources mentioned above, is summarized in Figures 5.2 and 5.3.

**Figure 5.2:** Flow of data in HDL-based synthesis and mapping

IP Cores can be again instantiated at direct request in the input code. There are, however, situations where a core generator or a similar internal procedure is called to implement a construct in the code.

Generators utilize relatively high-level functional specifications (e.g. “this is a signed multiplier”). Such an information must be either preserved from earlier design stages and taken over, or discovered. Both modes take place in practical HDL processing.

The net result for our discussion is that only combinational circuits for which there is no generator are interesting for (random) logic synthesis. The question is, what generators are available in a “usual” design system.

In general, the answer is “what the random logic synthesis and mapping cannot do well enough”. Generators are less flexible than general logic synthesis, and the effort to produce and verify them must be rewarded. When generators that produce physical implementations are used, the reason can often be summarized as “what the physical implementation cannot do well enough”. Yet another reason is verification – the generator is proven once, and generates many instances which are correct by construction. Moreover, a test for the block may be easy to
CHAPTER 5. LOGIC SYNTHESIS AND TECHNOLOGY MAPPING

Figure 5.3: Domains traversals in HDL-based synthesis and mapping; local optimizations omitted

produce for a specialized generator, but difficult for a general ATPG procedure. Multiple reasons are usually behind the inability to perform “well enough”. In the past, the need to utilize the target technology to the edge prevailed. Fast interfaces required for FPGA implementations (DDR RAM, fast serial interfaces) were problematic for both the logic synthesis and even physical design procedures. Similar problems were met in ASIC adder design; routing congestion in a Kogge-Stone adder (Kogge and Stone 1973) or optimum sizing of transistors in a Manchester carry chain (Pedroni 2008).

Some technologies, most notably FPGAs, have special features which are hard to employ by a general synthesis procedure. Hard-wired transceivers are provided to solve the serial interfaces problem; specialized carry chains improve adder performance (to the extent that carry-ripple adders are the most common kind in FPGAs; DSP blocks in FPGAs) will probably stay the target of logic generators and explicit instantiation for a long time.

The border case seems to be adders. They are ubiquitous in digital logic, yet simple enough to gradually become feasible for logic synthesis. As a result, multiple approaches to adder construction work-flow exist. In Figure 5.4a, the procedure used in full-custom circuit design is outlined. A generator composes the adder from special cells, connected by abutment to maximize performance and minimize area. As a result, the generator is specific to a single adder construction (ripple carry, carry look ahead, etc.) and to a single technology.

To get free of the technology restrictions, generators producing abstract gate netlists were constructed (Figure 5.4b). It is assumed that the underlying physical design procedures perform “well enough” and that the loss of performance caused by the design style used (mainly standard cells) is not significant.

In FPGAs, hard-wired carry chains are necessary for an acceptable performance. A proprietary FPGA design software can afford to be specific to a family of programmable devices.
Adders are then produced by a generator which “knows” the device structure (Figure 5.5a), and can produce a netlist of blocks existing in the device. The physical design procedure (also family-specific) then implements carry by specialized blocks and circuits.

Adders are not the only iterative structure that benefits from fast carry chains. To avoid a large number of generators in the design system, the random logic synthesis should perform “well enough” to produce an acceptable design of an adder or a comparator (Figure 5.5b). Such synthesis tools do exist (e.g. Amarú, Gaillardon, and De Micheli [2013]). The subsequent technology mapping procedure must be able to discover how to use structures existing in the device for implementation; it is assumed that it is family-specific at present.

Recent technology-mapping algorithms can discover how to employ a general heterogeneous programmable structure to implement a given circuit (Figure 5.5c). They are still computationally demanding and probably not yet in general use, but can bring substantial advantages for special circuits (Section 5.8 and Hu [2009]).

From this overview, it can be seen that the transition from Figure 5.5a to Figure 5.5b changes our definition of random logic synthesis. The design of adders and, to some extent, other iterative structures of similar complexity becomes a required capability for any logic synthesis. Although example circuits of this kind exist for a long time (S. Yang [1991b]), they are not sufficiently documented to tell what an acceptable solution is. Even if someone knew that the performance of academic tools on them was unsatisfactory (Fujita et al. [1993], page 123), this was never seen as a serious problem.

**5.4.4 Logic synthesis and technology mapping as an adaptive process**

Logic synthesis and technology mapping evolves constantly. This is not only the result of research, but also a consequence of evolving requirements. The most influential source of change is technology, which in turn defines the scale of circuits that can be produced. Competition builds up requirements on optimality. Designers, as the users of synthesis tools, wish to express their
ideas as concisely as possible, without absorbing algorithmic details of the tools first.

In the early eighties, there were technologies directly implementing a set of SOPs – PLA. The implementation consisted of two regular structures. The first one, called AND plane or AND matrix, produced a set of signals, each representing a term of the original SOP. The other structure, OR plane or OR matrix, summed up these terms to produce the desired outputs. The PLA layout depended only on the number of input variables, the number of terms and the number of outputs. The function itself was set up using a single mask in the case of an ASIC, or by programmable one-bit cells in the case of early field-programmable PAL or GAL devices. When PLA structures were considered acceptable no longer, multi-level synthesis appeared and slowly matured.

The scale of circuits was not very large in the PLA times. Optimality, however, was of prime interest, and the additional burden of specifying incompletely specified functions was put on the users. When larger scale forced the use of HDLs, multi-level synthesis procedures could not utilize flexibilities in the circuits well, and working with completely specified functions was considered easier from the designer’s point of view.

Since then, quality requirements increased, and the flexibility support in multi-level synthesis has been developed. DCs were adopted in design languages, sometimes in a makeshift and unofficial way (cf. Section 4.2). The design practice, however, is sluggish in adopting them, since it requires a change in thinking (Bečvár 2014).
5.5. TECHNOLOGY MAPPING

We documented the effort to characterize practical circuits in Section 4.1; we analyze its influence on algorithms evaluation in Chapter 6. Because we are unable to tell what a practical circuit is, synthesis algorithms and tools can be evaluated by benchmark circuits only; other means are not considered reliable. The benchmark sets thus shape synthesis procedures; synthesis adapts to benchmarks. Since the benchmarks (allegedly) reflects designers’ style and habits, we can say that synthesis and mapping procedures are social constructs.

5.5 Technology mapping

Technology mapping implements a given circuit with existing resources. The resources are at the gate level of granularity; each resource covers only a small part of the implemented circuit. All approaches to mapping comprise – more or less pronounced – the following steps:

Decomposition in a general sense, that is, finding parts of the circuit that could be implemented by a resource.

Matching of that part with existing resources, providing a candidate way to implement a part of the circuit.

Covering the circuit with resources, that is, selecting a set of candidate resources that optimally implements the circuit.

Methods used in each step depend mostly on the target environment, i.e., on the nature of its resources.

The simplest – and historically the oldest – situation is with gate libraries, such as libraries in the standard cells design style. A typical library contains gates realizing Boolean operators (such as NOR) or simple universal circuits (AND-OR-INVERT). The circuit and library gates are described as heavily restricted Boolean networks (e.g., as NAND networks). From those networks, structural matching based on subgraph isomorphism suggests candidate realizations (Keutzer 1987).

The most popular way to implement a programmable function is the LUT. It has a fixed number \( K \) of inputs (originally four, recently six and in special cases up to nine). Due to fully programmable content, a \( K \)-LUT can implement any function of \( K \) inputs, that is, any of \( 2^2^K \) functions. This number cannot be enumerated in a library. The ability to implement arbitrary function, however, reduces the problem to covering the circuit with parts that have up to \( K \) inputs and a single output – so called \( K \)-cuts (Section 3.1). Given the original circuit as a (restricted, fine-grained) Boolean network, \( K \)-cuts can be enumerated in the network. Cut enumeration can be restricted to a polynomial problem (Cong and Ding 1994); other approaches are possible (Francis, Rose, and Vranesic 1991). For recent devices with multiple outputs from a single LUT, \( KL \)-cuts are appropriate (Machado, Martins, Callegaro, R. Ribas, et al. 2013). When cut enumeration produces multiple alternatives to cover a node, the resulting instance of the covering problem must be solved. This kind of mapping algorithms is presented in most academic tools.

Few if any industrial FPGA devices are composed of uniform \( K \)-LUTs. The tables vary in size; they are combined in structures permitting to implement large functions; programmable carry chains implement iterative functions; fixed gates help with special functions such as multipliers. In other words, the devices are heterogeneous, so a more precise approach is required.

Consider implementing a multi-output Boolean function \( Y = F(X) \) using a programmable resource \( G \) having programming bits \( P \) so that \( Y = G(P, X) \). To find whether this is possible means solving (Ling, Singh, and S. Brown 2005):

\[
\exists P \forall X G(P, X) = F(X).
\] (5.1)
Chapter 5. Logic Synthesis and Technology Mapping

The matching is no longer structural. The problem is actually Quantified SAT\(_2\) (QSAT\(_2\)), which is known to be \(\Sigma^P_2\)-complete (Garey and Johnson 1979, Section 7.2). With simple devices such as Actel ACT 1 (Actel Corp. 1996), this problem was solved by exhaustive enumeration of \(P\) (Mailhot and Di Micheli 1993). Recent structures, however, require full QSAT. Suitable solvers were developed (Hu 2009; Safarpour et al. 2006; Cong and Minkovich 2007a).

The variants of mapping procedures are summarized in Figure 5.6.

---

5.6 Classical approach to logic synthesis and technology mapping

The first studied approach is classical in the sense that it understands its input as pure behavior, aiming to disregard any structure. As we will show, this is in general true, owing to efficient optimization of the behavioral description. From this starting point, structure is gradually introduced. To learn the properties of the procedures and their top-level control structure, we describe and discuss main algorithms of this approach.

5.6.1 Principle

Classical synthesis produces the resulting structural description from structured representation of the implemented Boolean functions, as described in Section 3.2 and Section 5.2.
When SOPs were implemented directly in silicon using PLA structures, the entire logic synthesis consisted of two-level multiple-output optimization (Figure 5.7a), and research focused on two-level minimization. The input was often prepared manually, and in that case the use of DCs was natural. A typical design system of this era, e.g. ABEL (ABEL 2003), consisted of FSM synthesis, two-level minimization, and device mapping.

SOP-implementing technologies, however, are only a detour in the development of technology. Implementations based on logic gates (with restricted number of inputs) connected in multiple layers have been common since the transition from contacts to vacuum tubes. They have substantial advantages in both area and performance (Hachtel and Somenzi 1996, Part IV).

Initially, a well-minimized SOP was considered to be a good starting point of multi-level synthesis (Rudell 1986; Hachtel and Somenzi 1996, page 10). Such a starting point was also, to a degree, independent of the form of original specification. Then, the logical step was to decompose the SOP form, to obtain a Boolean formula in factored form (Figure 5.7b). The structure of the factored formula is used as the structure of the circuit.

This approach to synthesis is actually the oldest one and dates back to Povarov (1954) and Ashenhurst (1959). Since then, it has been enhanced by a number of other restructuring procedures, such as extraction, substitution, extraction, and elimination, which will be described in the next section. As a consequence, it can produce a general Boolean network, not only a tree structure like a mere decomposition can. Also, the information on an incompletely specified function is no longer lost at the two-level minimization step. A study report by Perkowski and Grygiel 1995, which focuses on decomposition only, lists over 700 references.

At the same time, HDLs came into general use. They, or their synthesizable subsets, were originally not suitable for incompletely specified functions (Hassoun and Sasao 2002, page 84). From the perspective of synthesis algorithms, this move was an important, although rarely discussed change. Although the situation improved since then (Section 4.2), HDL designs use completely specified functions only. Incomplete specifications can be produced internally, e.g. from FSM synthesis.

On the other hand, human designers put structure into their descriptions, and that structure appeared to be also a good starting point of the multi-level synthesis, and also a comprehensible way to influence synthesis results (besides setting constraints). For all those reasons, the costly minimization procedure could have been discarded. If minimization was performed at all, it processed separate nodes of the Boolean network (Figure 5.7c).

This kind of synthesis has been incorporated into academic tools MIS (Brayton, Rudell, et al. 1987), MIS II and SIS (Sentovitch and al. 1992), and, allegedly, also into Synopsys Design Compiler (Hachtel and Somenzi 1996 page 441), one of the earliest industrial synthesis tools that gained broader acceptance.

Input and output to synthesis got either identical form (usually Boolean networks) or forms that could be made identical (extracting a Boolean network from a gate-level netlist). Thus, the synthesis could be repeated, which marks a paradigm shift towards resynthesis approach, covered in Section 5.8. Decomposition and other restructuring algorithms remain in focus, as they are useful building stones in other approaches to synthesis.

5.6.2 Algorithms of two-level optimization

Two-level minimization as introduced above operates on the SOP representation of the minimized function. A vector Boolean function \( F : B^n \rightarrow B^m \) can be seen as a set of \( n \) Boolean functions \( F_i : B^n \rightarrow B, i = 1 \ldots n \), which in turn is represented as a set of \( m \) SOP formulas of \( n \) variables.

To implement such a function, all implicants of all formulas must be covered. Depending on target technology, the cost of the implementation is the number of implicants (the size of a PLA
array) or the number of literals (the total count of inputs in a gate implementation).

An implicant chosen to the resulting set of SOPs contributes certain number of states to the on-sets of the functions $F_i, i = 1 \ldots m$. Therefore, the aim is to cover the on-sets with the smallest set of implicants, while avoiding all states of the off-sets. A naive minimization algorithm

- generates all implicants, and
- formulates and solves the covering problem.

To simplify both tasks, algorithms to find prime and essential implicants have been introduced. Even so, minimization algorithms still have exponential worst-case complexity, and heuristics must be used.

From mid-eighties, two-level minimization is dominated by ESPRESSO (Brayton, Hatchel, et al. 1982). The methods introduced or improved by ESPRESSO are numerous, so that an entire book on them has been written (Brayton et al. 1984). The heuristic is iterative. The input SOP is taken as the initial solution (cover). At each further step, ESPRESSO modifies the existing cover by applying the following operators to an implicant:
5.6. CLASSICAL APPROACH

Input expansion. Delete literals from the implicant, until it becomes prime or intersects with the off-set of some $F_i$. A heuristic is used to determine which literals to delete, i.e. which prime implicant from those covering the original implicant to produce. It increases the number of on-states covered by that implicant.

Input reduction. Add literals to the (originally prime) implicant while it still covers all states in the on-set of $F_i$, $i = 1 \ldots m$. It decreases the number of on-states covered by that implicant.

Output expansion. Add the implicant to a subset of functions from $F_i$, $i = 1 \ldots m$, until it intersects with the off-set of $F_i$. It increases the number of on-states covered by that implicant.

Output reduction. Remove the implicant from some functions from $F_i$, while it still covers all states in the on-set of $F_i$. If $F_i$ is the only function that uses the implicant, drop the implicant entirely. It decreases the number of on-states covered by that implicant.

After each action taken, redundant implicants, i.e. implicants entirely covered by other implicants, are removed from the cover. Notice that no instance of the covering problem is explicitly formulated or solved.

ESPRESSO is the de-facto standard for problem instances formulated by the designer, or by FSM synthesis. Outside this area, specialized minimizers exist and perform better on instances they are intended for. As an example, we describe shortly the BOOM family of minimizers.

The BOOM minimizer (Hlavička and Fíšer 2001; Fíšer and Hlavička 2003) has been constructed to deal with instances having thousands of primary inputs but a relatively small set of care implicants, such as the function met in Built-in Self Test (BIST) synthesis have. The algorithm uses a strategy (different from ESPRESSO) to find prime implicants for separate single-output functions, reduces them to find group implicants and then solves the covering problem. Later, FC-Min (Fíšer, Hlavička, and Kubátová 2003; Fíšer and Kubátová 2004b) used a different strategy to find group implicants directly, and therefore to process functions with a large number of outputs faster. These two approaches were united in the BOOM-II minimizer (Fíšer and Kubátová 2004a).

5.6.3 Algorithms of multi-level optimization

The process of transforming a SOP form to a multi-level equivalent description is traditionally called the decomposition of the SOP form. One such form describes a single-output function. To optimize multi-output functions, and to build general Boolean networks, other procedures are needed, typically comprising:

Decomposition. A Boolean function is expressed by two or more Boolean functions, with the assumption that those functions are collectively easier to implement than the original one. Factoring, often considered to be a special operation, is a special case.

Extraction. In a set of Boolean functions, a set of sub-functions is identified. By sharing their implementations, the implementation of the original set of functions shall be easier.

Substitution. A Boolean function is attempted to be expressed in terms of other, already existing function.

Collapsing or Elimination. An opposite operation to the above ones; expresses a Boolean function either completely as a SOP, or reduces the number of levels by absorbing other
functions. It can reduce not only the number of levels (delay), but also the size of the implementations in certain situations.

**Node minimization** Per-node optimization in the context of the entire Boolean network, which can induce additional flexibility.

**Decomposition**

Given an ordered set \( X = x_1, x_2, \ldots, x_n \) of Boolean variables, and a Boolean function \( F(X) \), the task is to express \( F \) as

\[
F(X) = H(X_0, G_1(X_1), G_2(X_2), \ldots, G_k(X_k)) \tag{5.2}
\]

where \( H, G_1, G_2, \ldots, G_k \) are Boolean functions, and \( X_0, X_1, \ldots, X_k \) are selections of \( X \).

Many decomposition methods start with finding a “good” partitioning first. This is known as the *partitioning problem* in decomposition. A test for decomposability is needed, which is the *decomposability problem*. When \( H \) is given, we get the special case of *decomposability wrt. specific function*. \( G_1, G_2, \ldots, G_k \) can be seen as separate Boolean functions, or the information they collectively transfer can be modeled as a multivalued variable. Any encoding can be chosen for that variable; as it influences the complexity of \( H \), the choice is known as the *encoding problem*.

Some special cases of decomposition are notable enough to be named:

**Disjoint decomposition.** All \( X_i, i = 0 \ldots k \), are pairwise disjoint. In the case of \( k = 1 \), \( X_0 \) is called the *free set*, \( X_1 \) is the *bound set*; \( G_1 \) is the *predecessor function*, \( H \) is the *successor function*. Most of decomposition methods have a disjoint and a non-disjoint version.

**Bi-decomposition.** \( X_0 = \emptyset, k = 2 \) (Sasao and Butler 1997). \( (F(X) \) is *groupable* (Bochmann, Dresig, and Steinbach 1991).

**Strong bi-decomposition.** \( X_0 = \emptyset , k = 2 \) as above, and \( X_1 \not\subseteq X_2, X_2 \not\subseteq X_1 \) (Mishchenko, Steinbach, and Perkowski 2001); \( (F(X) \) is *strongly groupable* (Bochmann, Dresig, and Steinbach 1991).

**Weak bi-decomposition.** \( X_0 = \emptyset , k = 2 \) as above, but \( X_1 \subseteq X_2 \) or \( X_2 \subseteq X_1 \) (Mishchenko, Steinbach, and Perkowski 2001); \( (F(X) \) is *weakly groupable* (Bochmann, Dresig, and Steinbach 1991).

**Gate-oriented decomposition.** \( H \) is given or selected from a limited set; also, \( k \) is usually limited. Perkowski and Grygiel (1995) calls this type simply *gate decomposition*.

A decomposition method can be characterized by the following features, as adapted from Perkowski, Luba, et al. (1995):

**Decomposition type**, i.e. a restriction of Equation 5.2 to a special case; \( k \) may be fixed, or \( H \) and \( G_i \) restricted to special form, etc. Notice that such restrictions are sources of bias, as they prevent decomposition from producing some circuits.

**Function representation**, i.e. the formalism used to capture actual state of the circuit, e.g. Boolean networks. The representation is closely related to techniques used to find candidate decompositions.

**The type of the function.** Besides ordinary Boolean functions, incompletely specified, multi-valued or even fuzzy functions can also be decomposed.

**The optimization criterion** used and the methods of its calculation.
Decomposition can be guided by the total variance of all signals. A circuit after functional decomposition has less variance than before. This approach is called functional decomposition and uses $\text{DFC}$ as the optimization criterion. For the definition and links to information theory, please refer to Section 4.1.1.

In the case of disjoint decomposition, a Karnaugh map of a function can be arranged so that it has the bound set values on one side (the horizontal one, by custom) while the free set values form the other one (vertical). The gain in decomposed function cardinality can be then calculated by counting equal columns of the map. Using this method (Ashenhurst 1959) the term column multiplicity is often used instead of decomposed function cardinality gain.

Decompositions which are not functional in the above sense use heuristic measures such as area or timing estimates, calculated from the underlying representation and according to the purpose of the decomposition. For example, decomposition used as FPGA technology mapping can count LUTs.

In the past, function representation and its manipulation techniques were of primary importance, and were used as the primary classification of decomposition methods into algebraic and Boolean methods.

An algebraic decomposition manipulates a formula. It does not assume that the formula is of a Boolean algebra, but only of some commutative and distributive algebra. Boolean methods, in contrast, use all axioms of Boolean algebra. The terms ‘algebraic’ and ‘Boolean’ are also used for method manipulating other representations, but in an equivalent manner.

Algebraic methods do restrict the state space of a decomposition algorithm and therefore introduce serious bias into the procedure. On the other hand, the restriction led to theoretical results (e.g. Brayton and McMullen 1982), which in turn led to efficient procedures.

At the prime time of algebraic methods, multilevel synthesis was considered very difficult. Hachtel and Somenzi (1996) thus introduce Part IV, Multilevel Logic Synthesis:

\textit{Compared to two-level logic synthesis, the problem of optimum multilevel logic synthesis is an impossible dream. [...] For the problem is an embarrassment of riches. Every where we look, we see an unending sea of don’t care conditions, impossible to fully enumerate, impossible to fully apply, yet fundamentally affecting the optimality of our results.}

Sentovitch and al. (1992) show a pragmatic stance in their explanation of extraction algorithms in the SIS system:

\textit{We do not know of any Boolean decomposition technique that performs well and is not computationally expensive; therefore we use algebraic techniques.}

With increasing computing power and greater demand for quality, however, Boolean methods did prevail. Quoting from Józwiak (1997):

\textit{Even the U.C. Berkeley group, which invented the algebraic factorization approach and was for a long time its most stubborn adherent, is recently devoting much attention to general decomposition [...]}. 

For comparison and analysis, manipulation methods are not of prime importance. As Perkowski and Grygiel (1995) put it in their analytical study of decomposition methods:

\textit{4. Functional Decomposition Versus Algebraic Methods.}
\textit{4.1 Decomposition is not ‘Algebraic’.}
\textit{Binary Decomposition is not a Boolean concept, it means, that it can be explained}
without using the concept of Boolean algebra and gates. What is necessary is only to describe a function (as a discrete mapping) in terms of composition of other functions. [...] The concept of decomposition can be explained without resorting to Post logic, Galois logic, or any other specific multiple-valued algebra to realize the functions.

During the last fifty years, numerous decomposition methods were developed, evaluated, and described.

Functional decomposition type has many sub-types and variants. Opinions on their relative importance vary, but the following sub-types represent different approaches:

**Ashenhurst decomposition.** $X_0 \neq \emptyset, k = 1$ (Povarov 1954, Ashenhurst 1959).

**Curtis decomposition.** $X_0 \neq \emptyset, k < | \bigcup k \{ k \} |$ (Curtis 1962). This is a generalization of Ashenhurst decomposition; $G_i, i = 1 \ldots k$ is understood as an encoding of a *multivalued function*. The above condition states positive gain in the DFC and functional nature of the decomposition. The method has been re-stated in different formalisms, for example Luba 1995, Ross, Noviskey, and Taylor 1991. Ross, Noviskey, and Taylor (1991) calls it *Ashenhurst-Curtis decomposition*.

**Roth-Karp decomposition.** A generalization of the above for incompletely specified functions (Karp 1963).

**PUB decomposition,** or Perkowski-Uong-Brown method in full (Perkowski and J. Brown 1988, Perkowski, Luba, et al. 1995 page 46). $H$ is a (conceptual) multiplexer. $X_0$ are its address signals which select one of $G_i$; the aim is to find at least some $G_i$ very simple, i.e. *trivial* in the original terminology.

**Information-based decomposition.** Jóźwiak (1997) built a general decomposition theory on the algebra of state machines by Hartmanis and R. Stearns (1966), stressing the links between decomposition and information theory. He also showed that a decomposition formalism can be based on the structural theory of partitions and set systems in Hartmanis and R. E. Stearns (1996). This approach has been tested in many areas of information systems; in the logic synthesis area, the applications are Jóźwiak and Volf 1992, Jóźwiak 1995, Jóźwiak, Volf, and Stevens 1995, Jóźwiak and Volf 1995, Jóźwiak and Volf 1995, Jóźwiak and Volf 1995, Jóźwiak and Konieczny 1996. The method allows for more general decomposition of combinational circuits, namely a structure where the blocks implementing $G_i$ can communicate.

**PLA decomposition of Sasao et al.** (Sasao 1989) has, unlike others of the time, $k > 1$ without using the multivalued approach. On the other hand, it is a disjoint decompositon.

Gate-oriented decomposition types offer the possibility to target a specific implementation environment, e.g. a gate library. They are used not only as logic synthesis methods, but also as first steps in technology mapping. Because they have practical orientation, some estimation of area or time is used as their optimization criterion. Some of the main directions are represented by the following sub-types:

**Factorization.** $k = 1$, and $H$ has a special form such that $F(X) = \sum_{x_i \in X_0} x_i + G_1(X_1).G_2(X_2)$, $G_1$ is the product of $x_i \in X_1$, and $G_2$ is the sum of $x_i \in X_2$. When applied recursively, such a decomposition produces *factored form* of $F$ and hence is called *factorization*. Because the condition $X_0 \neq \emptyset$ is not required, it is not a subset of Ashenhurst-Curtis decomposition; because $H$ is a sum, it can be subsumed under gate-oriented decompositions.

**Spectral methods.** Using e.g. Walsh transform (Karpovsky 1977, Hurst, Miller, and Muzio 1983), one can divide a Boolean function into a linear and a non-linear component. Thus, a linear pre- and post-processors of a circuit can be identified (Varma and Trachtenberg 1989).
5.6. **CLASSICAL APPROACH**

**XBOOLE decomposition.** (Bochmann, Dresig, and Steinbach 1991; Steinbach and Wereszczynski 1995; Mishchenko, Steinbach, and Perkowski 2001). This type is based on the notion of function groupability. It heuristically finds an initial feasible (that is, decomposable) grouping of variables, permitting a non-disjoint AND-, OR- or XOR-decomposition. Then it iteratively tries to put more variables into $X_1$ and $X_2$. The entire decomposition procedure is expressed in terms of complementation, cofactor, AND, OR, MIN, MAX and XOR. For those operations, the XBOOLE system (Steinbach 1992) is used, hence the name.

**BDS and followers.** The idea of identifying possible decomposition by structural situations in a BDD originated in Karplus (1988), was extended by C. Yang, Singhal, and Ciesielski (1999) and implemented in the BDD-Based Decomposition System (C. Yang and Ciesielski 2002). The indicating structure may be a single node (simple dominator) or a cut, a dividing line which crosses every path from root to terminals exactly once (generalized dominator). Dominators indicate AND, OR, and XOR decomposition. BDS can perform also two kinds of MUX decomposition.

**BDS-PGA** (Vemuri, Kalla, and Tessier 2002) changes the control algorithm of the decomposition. It applies the available decompositions until all decomposed function have less than $K$ input.

**BDS-MAJ** (Amarú, Gaillardon, and De Micheli 2013) adds decomposition with respect to the majority function; as such decomposition always exists, a mechanism to find proficient decompositions is designed.

**Postprocessing**

Decomposition, by its very principle, produces tree-structured networks. Usually, optimization deals with a multi-output function, but many decomposition methods treat the outputs separately. As a forest is in general not an optimal structure, further steps must be taken.

Common subfunctions can be identified in the optimized Boolean function, leading to a general structure. Such procedure is called extraction. It can be performed using algebraic methods as in the SIS commands gcx and gkx, or by BDDs (C. Yang and Ciesielski 2002).

Similarly, an existing subfunction may be used to implement another function (substitution). A subfunction may be removed from the original function, and the rest may be implemented in terms of another function. The term resubstitution is used for such a procedure.

All these procedures try to construct deeper and narrower structure. A reverse process is needed, for instance, to improve timing, or to escape a local optimization minimum. The process is called elimination, flattening, or collapsing. Collapsing is also needed to perform the classical optimization (which starts with two-level optimization) on a structured circuit.

**Minimization**

Decomposition and other circuit manipulation methods finally produce circuit description with structural component, in most cases a Boolean network or multi-valued network. Some behavioral component is still present, described by node functions. To implement the circuit, node minimization is necessary.

Two-level minimization is able to handle incompletely specified functions, and use the incompleteness as flexibility to produce better results. The context (environment) of a node function influences its flexibility. The surrounding network cannot produce some combination of node inputs, or transfer the output value to the primary outputs of the network.

Because the flexibility of a node function depends on context, it also changes with the context. When a node function is minimized using its flexibility, the flexibility of other nodes may be diminished. Several formalisms were proposed to handle this interdependency.
Don’t care conditions assume that the node function is constant, but incompletely specified. The unspecified values can originate from an incompletely specified function of the entire network (external DCs, EXDCs), or from the inability of the context to produce corresponding input conditions (satisfiability DCs, SDCs), or from inability to propagate node output (observability DCs, ODCs).

The flexibility interdependence complicates minimization to a large extent. Compatible ODCs (CODCs) are those which can be utilized simultaneously and have no interdependence. Methods to compute were proposed by Savoj and Bayton (1990); other contributions are Bartlett et al. 1988, Brayton 2001, Muroga et al. 1989.

Boolean relations were used for flexibility that cannot be expressed by DCs. A Boolean relation is commonly described by its characteristic functions, and that in turn by a BDD. Minimizers for Boolean relations were presented (Brayton and Soméazi 1989, A. Ghosh, Devadas, and Newton 1992, B. Lin and Soméazi 1990, Watanabe and Brayton 1993). Although working with Boolean relations requires greater effort than DCs, they are still not general enough. Multiple Boolean Relations (MBRs) (Sentovich, Singal, and Brayton 1993) have been shown to be the most general formalism for the purpose.

Sets of pairs of functions to be distinguished (SPFDs) are more general than ODCs but less than MBRs (Watanabe and Brayton 1994). A minimizer using this approach (Sinha and Brayton 1998) proved to be practical.

5.6.4 Decomposition and technology mapping

To enable structural matching, technology mapping requires decomposition of some kind. The classical approach using restricted Boolean networks with fine granularity (Keutzer 1987) has already been mentioned in Section 5.5. Dividing the design task between technology-independent and technology-dependent phase is serial task decomposition (Section 2.1) and as such is prone to quality loss.

Lehman et al. (1997) developed methods to direct algebraic decomposition to obtain networks that can be better covered with library circuits. Although it retains techniques used before, such as restricted Boolean networks and tree mapping, it is more effective.

A more recent example of technology-oriented decomposition is BDS-PGA (Vemuri, Kalla, and Tessier 2002, Section 5.6.3). With relatively small modification of the algorithm, gate-oriented decomposition of the original BDS has been turned into a LUT mapper.

5.6.5 Discussion of the classical approach

Algorithm architecture

Figure 5.8 shows the topmost control architecture in ESPRESSO. As the flow of control indicates, it is a local iterative heuristic. Procedures REDUCE and EXPAND realize expansion and reduction steps described above. The procedures themselves are iterative, controlled by a literal-choosing heuristic. The validity of the cover may not be violated, and therefore the outcome depends also on the order in which the expansions are done.

ESPRESSO minimizes the number of implicants. An expansion step per se does not change their number; an output reduction does so only if it deletes an implicant from all functions. These steps only make some implicants redundant and hence a candidate for removal. The illustrative examples in Hachtel and Soméazi (1996, Section 5.1.1) show that it is the interplay of expansion and reduction that makes the heuristic work. When an implicant is reduced, it is hoped that it will be covered by a neighboring implicant after a following expansion, or that it will be expanded in a different manner (Brayton et al. 1984, page 124).
Expansion and reduction forms a pair of operations which themselves are iterative, but should be interleaved during iteration at a higher level. That level is simple, applies the lower-level operation in a fixed order and is almost oblivious – it does not evaluate the circuit beyond the optimization criterion.

So far we have discussed the inner loop of the algorithm. The procedures REDUCE,GASP and EXPAND,GASP differ from their previous version in their control. They search for all feasible expansions and reductions first, and then they perform them all, without regard to their order. As a consequence, the cover may become infeasible. Together with IRREDUNDANT, they form a procedure called LAST,GASP. Brayton et al. [1984] describes its purpose thus:

This procedure is used as a final attempt to extract a few more cubes from the cover.

Rudell [1986] states its purpose in a similar way, yet in the pseudocode in Figure 4.1 we read:

/* Perturb solution to see if we can continue to iterate */
G ← LAST-GASP (F, D, R);

We can see that LAST,GASP serves both purposes. The outer loop of the algorithm can continue only because LAST,GASP changes the cover in a way that the inner loop could not. Moreover, two-level minimization is a problem where (sub)optimum solutions can be found at the border between feasible and infeasible configuration – removing any literal from an optimum cover would render it infeasible. Because the result of REDUCE,GASP is not necessarily a feasible cover, ESPRESSO adopts here a technique which later became known as strategic oscillations (Glover and Hao [2011]).
Figure 5.9 shows the top-level architecture of multi-level optimization in SIS. *script-algebraic* has been considered classical example of algebraic methods. In contrast to textbook explanation, it has a different order of steps. It begins with node minimization and resubstitution, and decomposition is the very last step.

Emphasis is given on extraction; it is always followed by resubstitution, forming another pair of interleaved iterative operations with mutual benefit. Again, the operations are performed obliviously and in a fixed order. Apparently, less beneficial changes in the circuits shall not prevent better changes from taking place, which is why the benefit threshold is gradually lowered, in a fixed number of steps.

**Independence of input specification**

*ESPRESSO* uses the input description as the initial cover. From that point, optimization proceeds iteratively. Therefore, the variance of input description is only *attenuated*, and potentially influences the output. The influence is, however, very small.

An example is in Figure 5.10. The s3271 circuit has been collapsed using the *collapse* command of *ABC*, and minimized using *ESPRESSO*. The number of literals in the input and output SOP were taken as a reference point. Other SOP descriptions of the same circuit were then obtained (for more details, refer to Section 8.4). They were also minimized by *ESPRESSO*. The number of literals in input and output in these runs are normalized to the reference pair of values.

The starting point of *BOOM* is always a tautology, and the input description is used as reference only (Hlavička and Fiser 2001). This screens the output from the influence of input variation. On the other hand, *BOOM* is a randomized algorithm of the Monte Carlo type, and the quality of output is a random variable. On 75% of the MCNC (S. Yang 1991a) benchmarks tested, no variation appeared. For the rest, Figure 5.11 is typical.

We see that two-level optimization provides an almost-canonical starting point for the entire optimization, which is then independent of input specification form. The circuit may be described...
in a very non-optimal way, and still the optimization possibly provides a reasonable solution. On the other hand, the structure created by a designer is lost; and the synthesis tools are not always able to rediscover it (Cong and Minkovich 2007b; Fišer and Schmidt 2009; Fišer and Schmidt 2012a).

### BDS architecture and bias

BDS (C. Yang and Ciesielski 2002; Yan, Ciesielski, and Singhal 2000) is one of the best-performing academic decomposition tools; only Bi-decomp seems to perform better (Mishchenko, Steinbach, and Perkowski 2001). BDS has been used in our experiments to obtain upper bounds of circuit size. Here its architecture and possible sources of bias are reviewed.

Decomposition algorithms are naturally recursive. Their top-level architecture is not iterative, but a global heuristic procedure in the sense of Servít and Zamazal (1995).

**BDS** uses BDDs to find candidate decompositions, as presented above. The entire input circuit can be represented by a single *global* BDD, discarding any input structure. To achieve scalability, per-node *local* BDDs can be used instead (C. Yang and Ciesielski 2000), building on the structure already present in the input. Global BDDs can be enforced and have been used in our measurements.

An important thought of the BDS’s authors is that a variable ordering which is good for BDD size is also good for finding dominators and hence good decompositions (Yan, Ciesielski, and Singhal 2000). This has been at least partly supported by experiments (C. Yang, Singhal, and Ciesielski 1999, Section IV.A); however, only reordering algorithms originally designed for BDD optimality (Somenzi 2012) were measured.

Candidate decompositions are tested in a fixed order by their efficiency (C. Yang and Ciesielski 2000) as summarized in Table 5.2. Once a candidate decomposition of a particular type is found, it is taken and the procedure continues at the next recursive level.
Figure 5.11: The distribution of output size produced by BOOM for the mainpla example. Notice the x-axis span. (Section 5.6.5, page 54)

Table 5.2: BDS decomposition types

<table>
<thead>
<tr>
<th>Decomposition</th>
<th>Gate</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple 0,1,x dominator</td>
<td>AND, OR, XOR</td>
<td>disjoint; algebraic</td>
</tr>
<tr>
<td>functional MUX</td>
<td>OR (MUX)</td>
<td></td>
</tr>
<tr>
<td>single MUX</td>
<td>OR (MUX)</td>
<td></td>
</tr>
<tr>
<td>generalized 0,1 dominator</td>
<td>AND, OR</td>
<td>disjoint; Boolean</td>
</tr>
<tr>
<td>generalized x-dominator</td>
<td>XOR</td>
<td>non-disjoint; Boolean</td>
</tr>
<tr>
<td>cofactor on top</td>
<td>OR</td>
<td>last resort</td>
</tr>
</tbody>
</table>

Decompositions with simple dominators are disjoint by definition. With generalized dominators, this is not necessarily the case. The BDDs used are ordered; the cuts used as dominators are always horizontal, and hence the bound set and the free set are disjoint.

Generalized x-dominator decomposition are taken even if the estimated size of the circuits increases by a limited amount, which, in rare cases, leads to an infinite recursion (C. Yang, Ciesieleki, and Schmidt 2014). Dominator-based decompositions prefer dominators in the middle of the BDD height to achieve better balance of the decomposition tree.

Scalability

The great benefit of the classical approach, the firm starting point, is also its Achilles’ heel. In practice, we often see very large SOPs, especially with XOR-intensive circuits. From theoretical point of view, this corresponds to the gap between AC⁰ and AC¹ circuits (Jukna 2012, Chapter 12). Thus, from a description of a yet manageable size we can get a description that is not feasible.

The complexity of most of synthesis algorithms is a function of description size, not just the number of inputs or a similar metric. Therefore, classical approach is not scalable.
Both the approaches presented below (which are, to some extent, related) aim to overcome the above mentioned limitations. They preserve the structure of the original input, and they avoid potentially infeasible transformations. The cost is, of course, their dependence on input specification form, and, hence, worse bias.

### 5.7 Rule-based approach to logic synthesis and technology mapping

Rule-based approach to logic synthesis is rarely mentioned in newer literature, as no recent procedures belong directly to this approach. Yet the approach is important for us to understand the behavior of resynthesis-based procedures, which share many features with rule-based systems (cf. Mishchenko and Brayton 2006).

#### 5.7.1 Principle

Expert systems were a promising application of artificial intelligence in late seventies and eighties. After their use in advisory roles, systems synthesizing a solutions emerged. Perhaps the most famous one is XCONN (Barker et al. 1989), a configuration system for Digital Equipment computers. The wave of enthusiasm for expert systems and other artificial intelligence techniques in engineering and especially in design is reflected in books such as Holden 1987; Gero 1988; Coyne et al. 1990.

A rule-based system can be thought of as an application of a knowledge engineering framework, that is, as a specialized expert system (Micheli 1994, pages 433–435). In the domain of logic synthesis and technology mapping, such a system usually consists of:

- a knowledge base of rules; a rule describes either
  - replacing a sub-circuit by another sub-circuit with the same functionality but better characteristics such as area or speed, or
  - the application of other rules (a metarule).
- an interpreting mechanism, often heuristic, that applies the rules to achieve a specified goal;
- a subsystem for knowledge acquisition and knowledge base maintenance.

An advantage of this approach is its flexibility. The software (knowledge base interpreter) stays unchanged; only the knowledge base can be changed and enhanced, possibly to accommodate “tricks” learned from designers, or rules for mapping into an unusual technology. Logic synthesis and technology mapping thus shared a unified approach (Micheli 1994, page 544).

This advantage, however, has also its downside. The knowledge bases were mostly unstructured sets of rules, hard to maintain and extend. The ordering of rules was misused to encode knowledge (Studer et al. 1999). The consequences of adding a new rule were difficult to predict and required detailed knowledge of the system’s inference engine. Thus only specialized knowledge engineers were able to utilize this advantage, and for most of design teams, the system was as closed as a compiled program. Moreover, the systems lacked scalability to cognitively complicated domains. As a result, rule-based systems shared the decline of interest in artificial intelligence.
5.7.2 Rule-based systems

One of the most cited rule-based systems is LSS by IBM (J. Darringer et al. 1984). It works in three domains:

- behavioral domain, oriented on AND/OR Boolean expressions, but also with higher-level entities such as parity generators or decoders;
- structural domain with abstract NAND/NOR blocks depending on target technology;
- structural domain with target library blocks.

This way, LSS covers both logic synthesis and mapping. The rules (called “transformations” in the paper) are of local nature, were designed manually and are considered a part of the system itself. Typically, they replace a sub-expression or a configuration of gates by another, functionally equivalent one. The designer can interactively run certain rules on a part of the design only, and is supported by analytical procedures, constant propagation, unconnected hardware removal, etc.

A different approach has been taken by SOCRATES (Geus and Cohen 1985). Logic synthesis is covered by classical modules (ESPRESSO two-level minimizer, algebraic decomposition, and technology mapping for AND/OR and MUX libraries). At the netlist level, rule-based optimization takes place. Rules are local transformations, similarly to LSS. The applicability of individual rules is determined by pattern matching, using a backtrack procedure. The top-level control strategy does not depend on rule ordering. The impact of a sequence of rules taken from partial search tree is estimated. Quoting form Geus and Cohen (1985):

> As noted above, optimization can be viewed as a state-space search problem. SOCRATES approximates a gradient search in traversing the search space, always applying the transformation that maximally decreases the cost function.

The partial search tree used is restricted by breadth, depth, and size. These parameters are not static, but change during optimization by a predefined schedule. Unlike LSS, designers are supposed to bring new rules, which is supported by a rule checking and loading application.

The optimization design by Farm, Dubrova, and Kuehlmann (2005) shares many features with the above described systems. It operates within the SIS (Sentovitch and al. 1992) environment, and shares its synthesis tools. One important difference is the set of transformations. They operate on AIG representation of the circuit, and their set of transformations is proven to be complete, ensuring reachability between any two states. Another important point is the control strategy, simulated annealing (Kirkpatrick, Jr, and Vecchi 1983). Moves that worsen the size of the circuit may be accepted, and the probability depends on size difference caused and on a globally manipulated parameter.

5.7.3 Discussion of the rule-based approach

The above described systems differ from mainstream expert systems in multiple ways. The designed artifact – the circuit – is complex, and efficient operation of such a system required an integrated model of the artifact (Wu et al. 1990), at the time a valuable contribution to design systems built on top of an ad-hoc collection of data so far.

From the algorithmic point of view, the local transformations established what Brayton, Rudell, et al. (1987) calls peephole optimization. Seven years (and five design systems) later, the Berkeley group also took this direction. These later systems, however, use mostly fixed transformations applicable to any sub-circuits, and are discussed in Section 5.8.

Rule-based systems became more and more similar to local optimization heuristics. SOCRATES approximates local search; the system by Farm, Dubrova, and Kuehlmann (2005) actually is local optimization. The left-hand side of a rule determines whether the right-hand side (the operator) can be applied to the given circuit (the state).
5.8. RESYNTHESIS APPROACH

Every local search needs a control strategy. When the state space is asymmetric so that the search cannot cycle, the control strategy can be primitive. Advanced heuristics with worsening moves need symmetric state space, and that in turn requires a more developed control strategy.

Already SOCRATES uses some ideas from simulated annealing besides the gradient search; the paper (Kirkpatrick, Jr, and Vecchi 1983) was highly topical at the time. Gradient search was common, as the heuristics had to converge rather rapidly, given the computational resources. Some of the examples used to evaluate SOCRATES are still in use, and rough comparison is possible. Assuming that a ‘transistor pair’ in SOCRATES report corresponds to a ‘literal’ reported by SIS, we find that the performance of SOCRATES is still comparable to e.g. BDS (C. Yang and Ciesielski 2002). The run times are two to three orders of magnitude longer, but on hardware which is five orders slower than recent workstations. It suggests that the carefully designed control strategy does bring advantage.

5.8 Resynthesis approach to logic synthesis and technology mapping

The scalability problems with the classical approach and the knowledge representation problems of rule-based systems led to a new approach, which combines the strong points of both its predecessors. It is algorithmic like the classical approach, but processes only a part of the circuit at a time, like the rule-based “peephole optimization”.

5.8.1 Principle

Resynthesis (Mishchenko and Brayton 2006) operates locally, on a part of the circuit only, as the rule-based systems do, but algorithmically, as in the classical approach. Moreover, as the local algorithms are fast, many passes of the local transformations can be done. Uniform representation of the circuit is used, which allows to combine synthesis steps in any order.

Unrestricted Boolean networks have too many degrees of freedom, which are difficult to utilize. Restricted networks can be auto-compacted (Bjesse and Boralv 2004), that is, restriction on redundancy can be enforced during construction of the network. Examples of such networks are Boolean Expression Diagrams (BEDs) (Andersen and Hulgaard 1997), Reduced Boolean Circuits (RBCs) (Abdulla, Bjesse, and Eén 2000) and AIGs (Kuehlmann and Krohm 1997, Kuehlmann, M. Ganai, and Paruthi 2001). The use of AIGs prevails in recent systems, as it is the simplest representation.


FRAIGs are semi-canonical; no two nodes in a FRAIG represent the same Boolean function in terms of the primary inputs. The same function may have different FRAIGs when e.g. the nodes are constructed in a different order.

The original “peephole” (Section 5.7.3) evolved into problem-oriented selection techniques, which can be divided into cuts and windows.

K-feasible cuts (Section 3.1.4, page 16) were originally developed to represent a part of the network which could be implemented by a single LUT (Pan and C.-C. Lin 1998). The idea of factor cuts (Chatterjee, Mishchenko, and Brayton 2000) brought an efficient way to enumerate K-feasible cuts, replacing the former FlowMap-like methods (Cong and Ding 1994). This enabled to use K-cuts for every local transformation which would profit from limited number of inputs.
KL-feasible cuts (Martinello et al. 2010) were proposed to represent multi-output blocks in FPGA mapping. Machado, Martins, Callegaro, R. P. Ribas, et al. (2012) and Machado, Martins, Callegaro, R. Ribas, et al. (2013) transferred the idea to mapped circuits (i.e. general netlists) and used in library-oriented technology mapping.

The technique of windowing originates from Mishchenko and Brayton (2005). A window of a node \( N \) is characterized by two integers, \( m \) and \( n \). First, the \((m + n)\)-distance transitive fan-out of all nodes in \( m \)-distance transitive fan-in of \( N \) is determined. Then, a vice versa, the \((m + n)\)-distance transitive fan-in of all nodes in \( n \)-distance transitive fan-out of \( N \) is also determined. The intersection of the sets is the window. Thus, a window is the context of the node \( N \).

### 5.8.2 Main algorithms

Mishchenko and Brayton (2006) identifies the following basic algorithms: rewriting, resubstitution, balancing, and redundancy removal. We outline the main algorithms here for the reader to understand what happens when such algorithms have to produce a circuit structure fundamentally different from their input, or when they are iterated.

**Rewriting**

Rewriting is, in fact, another name for rule application in rule-based systems, and their only operation. Rewriting can serve many purposes; optimization for area, for speed, or even technology mapping.

Rewriting as implemented in the ABC system (Mishchenko, Chatterjee, and Brayton 2006) is based on Bjesse and Boralev (2004) and works on \( K \)-cuts, originally for \( K = 4 \). For such a small value, the number of NPN classes is still manageable, and optimum implementations of the circuits can be precomputed and stored. It is interesting that not all NPN classes are actually used; some allegedly do not appear in designs.

This kind of rewriting led to the idea that harvesting, storing and reusing existing circuit implementations (from any tools) can be sufficient as a synthesis and mapping procedure (W. Yang, L. Wang, and Mishchenko 2012). The techniques for storage and retrieval were later based on the fact that a canonical form can be constructed from disjoint decomposition (Mishchenko and Brayton 2013). Besides canonicity, this form preserves properties important for synthesis, such as symmetry and decomposability.

The reuse of an implementation is in principle identical to technology mapping. Therefore, it requires some kind of matching. In the discussed procedures, fast NPN classification (Huang et al. 2013) was used.

**Refactoring, resubstitution and balancing**

Almost any existing synthesis procedure can be transferred to resynthesis. One example is classical algebraic refactoring (Brayton and McMullen 1982) on windows, resubstitution and path length balancing (Cortadella 2003).

Flexibility-based resynthesis (Mishchenko, Brayton, J.-H. R. Jiang, et al. 2011) (the \texttt{mfs} command in ABC) uses two windows like Mishchenko, Zhang, et al. (2006). The inner one is the optimized part of the circuit; the outer one provides context, where the external DCs are derived as in Case, Mishchenko, and Brayton (2008). Satisfiability and interpolation (McMillan 2003) is then used to optimize the inner window content.
5.8. RESYNTHESIS APPROACH

Choosing

The idea for alternative implementations for some network nodes originated in technology mapping (Lehman et al. [1997]) and has been generalized for resynthesis as *choicing* (Chatterjee, Mishchenko, and Brayton [2006]).

Technology mapping algorithms

With efficient \(K\)-cut enumeration, homogeneous LUTs mapping is relatively straightforward (Mishchenko, Chatterjee, and Brayton [2007]). For each node, \(K\)-feasible cuts are enumerated and stored. Then, for each node the cut giving minimum path length is determined (depth-oriented mapping). The cut is updated to save area in several passes (area recovery). Later on (Mishchenko, Sungmin, et al. [2007]), the idea of cut evaluation and sorting was introduced, leading to *priority cuts*. This is the `if` command in \(ABC\).

The techniques of storing, matching and reusing implementations were used to construct a mapper for heterogeneous target FPGA architectures (Mishchenko, Brayton, Feng, et al. [2015]). As NPN classification is insufficient for the task, a dedicated Quantified Boolean Formula (QBF) solver has been used.

5.8.3 Algorithm architecture

Synthesis in \(ABC\) is usually controlled by scripts, such as `resyn` or `choice`. Each script invokes a sequence of \(ABC\) commands such as `rewrite`. The `dch` command that replaced the scripts and has been used in our experiments is not a script, but has a similar purpose and position. Figure 5.12 outlines its structure. The role of choicing is apparent. The branches of the choice

![Figure 5.12: High-level architecture of the `dch` command in `ABC`. Procedures in shaded boxes accept changes even when the gain is zero.](image-url)
have fixed sequences of ABC commands. Rewriting is interleaved with refactoring. These transformations differ in nature – rewriting operates on cuts, while refactoring uses windows; rewriting inserts precomputed circuits, refactoring computes decomposition. Therefore, the interleaving may lead to better functioning of the algorithm. Quoting from Berkeley Logic Synthesis and Verification Group (2000, Section “Combinational synthesis”):

*It can be experimentally shown that iterating these two transformations and interleaving them with AIG balancing (command balance; standard alias b) substantially reduces the AIG size and tends to reduce the number of AIG levels.*

Zero-gain replacements are permitted only in final stages of both sequences, permitting less beneficial transformation to take place only after more proficient ones had a chance. With the exception of the final choosing, control is oblivious.

A typical construction of AIG optimization algorithm can be seen. Nodes are traversed in topological order. At each node, cuts or windows are enumerated and processed.

The `if` command in ABC is an example of technology mapping algorithm, “if” standing for “interface” (to a mapping package). The command can work as area-oriented or delay-oriented. Both modes of operation start with delay-oriented mapping. Afterward, area is “recovered” (in the original terminology) when required (Figure 5.13).

![Figure 5.13: High-level architecture of the if command in ABC](image)

*Area flow* is an optimization criterion which captures the usefulness of a node’s implementation in the current mapping. Area flow of a node increases with the area needed to implement the selected cut of the node and with the area flow of the input nodes of the cut, but decreases with the fan-out of the node.

Typical patterns of algorithm design can be observed: oblivious and fixed top-level architecture, topological ordering of nodes within optimization passes, and gradual change in optimization criteria, as was shown in two-level and multi-level optimization. At the current stage of analysis, it is not clear whether this architecture is common to all such systems, or is characteristic to the Berkeley group only. The systems discussed, however, were influential in their respective periods.
5.8. RESYNTHESIS APPROACH

5.8.4 Additional layers of iteration

The fixed top-level architecture of ABC resynthesis has been optimized to perform well on most benchmarks, and to consume reasonable time for practical purposes. In the problematic cases studied here and for estimating good upper bounds, however, we are interested in maximum performance. Then, it is reasonable to investigate whether the existing control utilizes the optimization potential well.

Creating an additional layer of iteration on top of the entire run of logic synthesis and technology mapping was considered by the Berkeley group itself. Quoting form Berkeley Logic Synthesis and Verification Group (2000, Section “LUT-mapping”):

"Sometimes significant area improvements without delay degradation can be achieved by repeatedly running choice; fpga; ps. Typically it takes more than 10 iterations to converge and the area keeps improving. This is a confirmation of (a) the ability of AIG rewriting to find good circuit structures, (b) the ability of choices to capture structural flexibilities, and (c) the ability of the mapper to do a good area recovery." choice is a predecessor of dch, fpga is a LUT-oriented mapper. The technology mapping steps apparently disturb the structure of the circuit, and prevent the iteration from premature convergence. Another example is the evaluation of the mfs command (Mishchenko, Brayton, J.-H. R. Jiang, et al. 2011). Quoting from that source:

"Section mfs [of a result table] corresponds to four iterations of technology mapping with structural choices, interleaved with the proposed resynthesis (st; dch; if {C 12; mfs {W 4 {M 5000})."

One way of preventing a local search to converge prematurely is to inject randomness. Fišer (2012, Chapter 4) uses the input text bias present in the procedures for this purpose. The procedure reacts mostly in the expected way, by slower convergence and better results. Fišer (2012) documents that this is not always the case. Still, the procedure is not guaranteed to asymptotically converge as was documented in (Fišer and Schmidt 2012c) on the alu4 example (McElvain 1993).

5.8.5 Discussion of the resynthesis approach

Quoting from Mishchenko, Sungmin, et al. (2007):

"However, it was found experimentally that restricting the order to a topological one is advantageous for most benchmarks."

This is the approach to synthesis procedures that advanced the state-of-the-art of industrial synthesis considerably, and formed a tradition of creative, innovative and serendipitous tools construction. For those plodding behind in an attempt for principled understanding, it is quite hard to comprehend. The performance problems, which are the reason of this study, certainly are not “most benchmarks”. Understanding the behavior and performance of resynthesis algorithms would help us to find out whether the problems are typical.

Additional iteration has been extensively studied in Fišer (2012 Chapter 3), Fišer and Schmidt (2012b), and Fišer and Schmidt (2012c). It appears that the results keeps improving after several thousand iterations in certain cases, but gets stuck in a local optimum in others (Fišer and Schmidt 2012c). The convergence curves demonstrate that a better convergence control is needed.
Randomness injection does improve the situation. In certain cases, the resynthesis procedures show a tendency to premature convergence. Randomness injection is, of course, a crude mean to control convergence. And even in the case where the results keep improving for a great number of steps (and impractical computing effort), we still do not have enough performance to handle examples such as the LGSynth'93 version of \textit{alu4}.

This measurement did not, however, consider the true nature of that version of the \textit{alu4} benchmark circuit. As presented in Section 4.3, this example is a part of the LGSynth'93 set (McElvain 1993), where all multilevel circuits were subjected to very substandard synthesis, and as a consequence contain much redundancy. Such an example can no longer be considered practical and relevant for synthesis procedure evaluation.

5.9 Summary

- Logic synthesis transforms behavioral description of a circuit into structural domain.

- Certain parts of a HDL circuit description such as FSMs or arithmetic are synthesized by specialized procedures called generators or parametric cores; the rest is processed by random logic synthesis. The boundary is not clear and not very well documented, thus the requirements for random logic synthesis are rather vague.

- Logic synthesis and technology mapping is in the process of constant adaptation to evolving technology, optimality requirements, algorithm research, and design style.

- The adaptation process is mediated by benchmark circuits, which represent how the designers think; any more insightful representation is not available.

- Technology mapping transforms structural description into another structure built from directly implementable blocks.

- The target environment for technology mapping can be described by enumeration of library blocks, as a homogeneous structure of LUTs, or as a general structure of programmable blocks. Structural matching and covering suffices the first two cases, while QBF solvers are necessary for the last.

- Synthesis and mapping procedures are heuristic in nature, and therefore prone to prefer some type of solutions over the others – a bias. A procedure’s bias may be caused by circuit representation, heuristic operators used, the inability to span long distances in the state space, by external requirements, implementation issues, or randomization.

- The classical approach to logic synthesis and technology mapping starts with two-level minimization followed by decomposition and (mostly structural) technology mapping.

- The two-level minimizer \textit{ESPResso} is a de facto standard. It has small bias, and provides a stable initial state for synthesis, independent of input description size and style.

- Multi-level synthesis consists of decomposition, postprocessing, and node minimization.

- Approaches to decomposition differ by the optimization criterion used (functional decomposition requires the decomposed blocks to diminish variance), by admissible blocks in the decomposition (e.g. AND decomposition) and in techniques (algebraic versus Boolean techniques).
Postprocessing identifies or creates common subcircuits; extraction and (re-)substitution are examples of such procedures.

Node minimization employs flexibilities imposed from outside or existing within the circuit network.

The classical approach pays for the stability of minimum SOP by its size, and therefore is difficult to scale.

Rule-based systems brought the concept of local transformation and added flexibility of the knowledge base; yet shared the problems of knowledge structuring and acquisition with other expert systems.

The resynthesis approach is distinguished by

- simple circuit representation,
- algorithms that enumerate and select parts of the circuits (cuts, windows),
- algorithms that profit from the limited size of a cut or window, and
- iterative operation.

Operations known from rule-based systems (rewriting) or from the classical approach (refactoring, resubstitution, balancing, etc.) were ported to this environment.

The algorithms studied share the following patterns of algorithm design:

- oblivious and fixed top-level architecture;
- gradual change in optimization criteria and/or thresholds;
- topological ordering of nodes within optimization passes.

Traversal in topological order is characteristic for the resynthesis approach transformation.

Iterative use of entire resynthesis procedures is possible. Technology mapping serves the role of solution disturbance.

In certain cases the solution keeps improving after several thousands iterations, in others reaches a minimum much sooner.

The bias of synthesis procedures can be utilized to inject randomness into such an iteration. It slows down convergence and in many cases improves final results.

Even under extreme conditions, such a procedure is not guaranteed to asymptotically converge.

An improved top-level algorithm architecture with a better-informed convergence control is a promising way to improve performance.
Chapter 6

Experimental evaluation

EDA tools are artificial, engineered objects entirely known to their authors. Intuitively, analysis of such objects should be able to reach any desired depth. Very often, we cannot use this advantage. The procedures comprising EDA tools are heuristic in nature, with no guarantee of performance. The heuristics themselves are layered to cope with the complexity of their respective tasks, i.e., there are heuristics within heuristics. Only loose upper bounds, often exponential, of complexity can be established and these are of little importance to both practice and understanding. The main input – practical circuits – is difficult to characterize as a subclass within the bigger class of Boolean functions (cf. Section 4.1), and therefore relevant statistical assumptions are hard to find.

The only existing escape from these difficulties is to admit that the procedures are too complex to be seen “form inside”, and to apply methods developed in science for such complex objects, describing them “from outside”.

This chapter analyzes evaluation procedures with respect to peculiarities of the EDA field, and aims to leverage them so that the results can serve as a source of understanding. The primary source and motivation for this chapter is our participation in work that relies on such methods heavily (Servít and Schmidt 1980). Experimental algorithmics, a very close field, is surveyed in Hoos and Stützle (2007). Guba and Lincoln (2005) helped to realize that the questions we ask are partly qualitative, and that this fact may be controversial. The ESPRESSO MV Report (Rudell 1986) is a nice example from the days when EDA procedures were simpler and had more theoretical investigations in the background.

6.1 Principle and origin

In this respect, we are close to the field of experimental algorithmics (Moret 2002; Hoos and Stützle 2007). The ultimate goal of this discipline is to bring scientific standards of reproducibility to the study of algorithms, and also to refute that experimental methods are the antithesis of theory. Experimental algorithmics has a much broader scope than heuristic optimization, which is the base of most EDA tools. Its main focus is on algorithms where the input has a relatively simple structure. Moreover, most studies in this field measures, characterizes and predicts complexity only in the form of run time or, more generally, cost of search.

Neither the term experimental nor the alternatively used term empirical has the correct set of connotations. “Experimental” can mean “by trial and error”, while “empirical” may also mean “without theoretical background”, which both do not apply to the field (Moret 2002).
The object of our interest, however, is more complex and difficult. We study an entire tool (application) instead of an algorithm; we have to use practical instances, for which neither empirical (Hooker 1994) nor mechanistic (McGeoch 1996) models are known (Chapter 4). In the field of EDA tools, a mere 5% improvement in performance is often worth a paper. On the other hand, we ask the experiments sometimes to offer insight into the algorithms, which needs even more precise measurement and interpretation. Therefore, we need high-quality evaluation procedures, as established in science.

Natural science deals with complex objects that can be analyzed only to a certain depth, and often with limited accuracy. Therefore, experiments are the primary means, and theory follows, providing explanations of experimental observations. This is an important part of the scientific method and has been refined and discussed for a long time. For a detailed explanation, we refer the reader to the book (Fisher 1993).

The general setup of a science experiment is in Figure 6.1. To design an experiment, it is crucial to realize what do we want to know; that is, to what question we seek an answer. From the question, the plan of the experiment follows. The object of study is placed in a designed and controlled environment, exposed to chosen stimuli, and its response is observed.

The results must be interpreted to be useful. The experiment tells us about the behavior of a concrete object under concrete circumstances. The questions, however, are about the behavior of a class of objects, and to answer them, the observations must be generalized. This is considered the most difficult and questionable step in an experiment, and the entire experiment design must support it.

![Figure 6.1: The paradigm of experimental evaluation](image)

### 6.1.1 Randomization, replication, and organization

In science, stimuli and responses are quantitative and therefore measurable variables (Figure 6.2). Often, the experiment is planned to reveal how a response variable, say, \( r_j \), depends on an input variable, here \( s_i \). Using multiple experiments and observing multiple dependencies, understanding is gradually achieved, as the ultimate goal of science.

![Figure 6.2: A quantitative experiment](image)

The setting and measurement of physical variables is never free from errors; both the set
stimuli and measured responses are imprecise, and possibly varying when repeating the experiment. Usually, we can make plausible assumptions about the probabilistic distribution of such a variance. When the experiment is performed multiple times (replicated), our knowledge of the distributions enables us to use statistical methods to suppress the errors. This principle was formalized as **randomization, replication, and organization to reduce error** by Fisher (1993).

Although this method is probably the best we can use, it does have some pitfalls. First, the usual assumption about the error distribution is that its mean value is zero; often, also that the distribution is Gaussian. With physical processes, these are probably safe assumptions; with others, they may not be. Moreover, the design of replication assumes that all influences are known. History of science shows that such an assumption may not be true.

### 6.1.2 An algorithm as a black box

Note that in this classical view, the object of our experiment is seen as constant, and its behavior as immutable by any input. Therefore, the obtained description of the object behavior, that is, the quantitative dependence of its response on stimuli, has a deterministic, not probabilistic nature.

A procedure or an algorithm is but a special case of an experimentally studied object (Figure 6.4). One can ask why an artificial object like an algorithm must be studied experimentally. There are several reasons, some connected with the nature of the algorithms, some with their purpose.

The first reason is that we are mostly interested in expected values rather than in a worst-case analysis. Worst-case situations are infrequent and in many cases can be tolerated in practice.
Probabilistic analysis has been performed for relatively simple algorithms only, and often, the results cannot be applied directly.

Logic synthesis is a discrete, combinatorial problem. Current combinatorial optimization algorithms are not able to solve this task as a monolithic problem, and task decomposition as described in Chapter 2 must be employed. The decomposition leads, in turn, to a complicated, layered algorithm architecture. The behavior of such a structure cannot be studied analytically. From this point of view, the object of study – the algorithm – becomes as complex as a natural object and we have to resort to the same methods.

Another reason – and perhaps the most intriguing one, as will be shown later – is the nature of the input. In Chapter 4, we show that inputs relevant to practice cannot be characterized in any way, not only in a manner useful for probabilistic analysis.

6.2 Questions, answers and requirements

In natural science, an experimentally answered question usually concerns quantitative measures of both the stimuli and the response in the experiment, with the intention to derive a formula that binds these quantities and can serve for further prediction.

For example, the influence of ambient light wavelength on plant growth is to be studied, to enable growing plants in an artificial environment. The plants are put into a controlled environment, where every known influence, such as temperature, humidity, and carbon dioxide...
content is held constant. Only the wavelength of the light is varied, and weight gain in the plants is observed. After filtering out, e.g., measuring errors, the required quantitative model can be constructed. For the model to be valid and accepted, the generalization must be trustworthy. For example, the set of plants used in the experiment must be statistically representative for the given species in growth, metabolic rate, chlorophyll development, and so on. If, later on, it is revealed that the experimental plants came from a non-standard subspecies, the whole experiment must be discarded. The same happens when some uncontrolled factor, e.g. the salinity of water, is discovered.

Such an experiment provides us with an external view of the process. If we wanted to understand it, we would have to study the functioning of chlorophyll, and nutrition transport in plants. Such a study would be supported by a number of experiments, but these would be targeted to explanation rather than quantitative modeling.

The difference between the external and internal view of events, and the potential of the latter to bring important knowledge, forced, among others, social science to accept qualitative research as a valid method. Quantitative methods have been refined for hundreds of years, and are thus seen as trustworthy, or “scientific”, which qualitative methods succeed to match only in a small part.

To bring qualitative methods into engineering would be judged as cheek. Engineering methods are often empirical and heuristic. In the privacy of their cubicles, engineers would admit that to replace a small portion of the empiricism by understanding would help. This work tries to enhance our understanding how logic synthesis algorithms work. Because of such a direction, it has to be partly qualitative, and we will pay attention to experimental methods aiming at qualitative results.

Whether a research is qualitative or qualitative can be recognized by the questions it asks. As the above figures show, the entire experiment planning depends on the nature of the question answered. We review the influence of questions on the experiments.

### 6.2.1 Practical performance prediction

The question, how the proposed algorithm will perform in practice, is the basic impulse to algorithm evaluation. The relevance to practical use is the most important factor. Because of the above mentioned generalization, the experimenter has to prove that all settings of the experiment are representative for practice. This includes not only circuits, but computing resources and metrics used in the experiment. Hooker (1994) mentions prediction as one of the main goals of experimental algorithmics.

This is the prevailing kind of experiment performed in the EDA industry. The reasons for it are connected mainly with intellectual property protection. First, in the EDA ecosystem, academia provides and publishes algorithms, which the industry then implements to industrial quality software. Therefore, the real environment and implementation details are known to the industry only. Second, nearly all practical circuit examples are under protection. The consequences are studied in Section 4.1. Finally, the industry surveys the users’ needs and preferences with various marketing methods, whose outcome is strongly connected to particular software and as such is not published.

To summarize, realistic circuits, environment and metrics are crucial to answer this class of questions.
6.2. QUESTIONS, ANSWERS AND REQUIREMENTS

6.2.2 Algorithm demonstration and comparison

Academics proposing a new algorithm have to prove its usefulness, or their work is ignored. Thus, it is the most common mode of experimental algorithmics. Methods to compare algorithms even with additional sources of variance, e.g., randomized algorithms, were developed (Hoos and Stützle 2000; Hoos and Stützle 2007), but still are not the norm in the EDA field, despite efforts such as Shum and Anderson (2012).

In most cases of EDA algorithms, full practical performance prediction as outlined above is not feasible, and mostly not necessary. The need for realistic setting is still present but less strict. The algorithms are evaluated relatively to the best known algorithms for the given task, which calls for standardized settings. Standard circuits are provided as benchmarks. More precisely, they should be called benchmark circuits, as they are but a part of an experimental setting. The rest is difficult to standardize; computing platforms evolve, and metrics change with advancing technology.

For this class of question, standardized circuits are required to permit comparison. The rest of the setting should be at least reproducible. At the same time, practical relevance of the circuits and metrics must not deteriorate too much.

6.2.3 Qualitative understanding

Once we ask why the examined algorithm does (or, so often, does not) work well, the outside view of the quantitative evaluation does not suffice. Hooker (1994) argues that empirical studies and theory in fact support each other, and that makes the results of experiments qualitative in nature, despite their quantitative cloak. For example, Reeves (1998) and Watson, Whitely, and Howe (2005) use quantitative observations to understand how the state space of a combinatorial problem influences the working of a local search procedure.

EDA algorithms tend to be sensitive to more subtle characteristics than just the instance size. These characteristics may be known by algorithm design (many experimental tools based on ROBDDs preserve variable ordering present in the input file) or have to be discovered (Fišer and Schmidt 2012b). That may call for a crafted input. Artificial (synthetic) input may tell us much about the algorithm. Finally, however, we must ask whether the information is relevant for a practical use of the algorithm, and we come to the relevance requirement again. Instead of a crafted input, we prefer a realistic input with desired characteristics suitable to answer our question. Such characteristics may themselves be qualitative in nature, and have to be determined by analytical procedures.

The setting for qualitative experiments can become more complicated (Figure 6.6). The base for experimental input data are still realistic examples. For the purpose of the experiment, they are analyzed, measured, classified and transformed. The transformations themselves may or may not be based on real design procedures.

Any experiment can be thought of as composed of basic steps. At each step, the properties of the input must be established, a procedure run, and its output measured (Figure 6.7). Many procedures, either transformations or the evaluated procedures themselves, provide additional information about its input and/or output. In that case, the measuring procedures became parts of other procedures.

The objects called “input” and “response” in Figure 6.7 can vary in nature. When evaluating a logic synthesis procedure, it is natural that “input” is a representation of a circuit and “response” is a representation of a functionally equivalent but optimized circuit. For example, an ATPG procedure takes a circuit representation as input and produces a test as a response, and so on.

From this point of view, the experimental settings in Figures 6.4 and 6.5 can be seen as oriented bipartite graphs consisting of data (representations of some objects) and procedures,
CHAPTER 6. EXPERIMENTAL EVALUATION

Figure 6.6: Using transformations and measurement procedures in experiments

where edges represent data flow. Both data vertices and procedure vertices are labeled with problem-specific attributes.

The need to eliminate variance (suppress noise) may be much greater for answering qualitative questions than in other situations, as higher levels of confidence may be required to establish qualitative findings.

Many examples of such “threshold” effects exist; one is explained in D. L. Goodstein and J. R. Goodstein (1996). In astronomy predating Tycho Brahe, an accuracy of 10 arcminutes was common. Tycho improved the accuracy to 2 arcminutes around 1580; the communications people today would say that he managed a 7dB noise reduction. Until Tycho’s measurements, the theory of circular trajectories of planets was consistent with available data. With Tychonic data, this was no longer possible, and led Kepler to suggest elliptic trajectories in 1609. Together with the rest of Kepler’s laws, this led to Newtonian physics in 1687.

6.3 Optimum solutions and bounds

In combinatorial optimization, the quality of a solution is measured relatively to the optimum quality (Aussiello et al. 1999). This is essential in establishing the notion of approximative algorithm.

In logic synthesis, this has a different but equally important role. Later on, we will demonstrate a circuit for which an average ABC result has 2625 gates. After investing 100 times more time, the solution improved to 1905 gates. This could be an impressive result, if Cartesian Genetic Programming (CGP) (Fišer, Schmidt, Vášiček, et al. 2010) had not provided a solution with only 73 gates. It seems to put the previous results into a proper perspective. However, even this result is not guaranteed to be optimal; it merely establishes an upper bound on the optimum
6.4. SPECIAL REPLICATION REQUIREMENTS

As we show in the next section, there are steps of logic synthesis, for which optimum results can be obtained. Cong and Minkowich used for such examples the term Logic Example with Known Optimum (LEKO). Where only an upper bound (for a minimization problem) is known, the term Logic Example with Known Upper bound (LEKU) was used. Notice that Cong and Minkowich had the decency not to call the examples benchmarks. The terms LEKO and LEKU have not come into general use; nevertheless, they denote important classes of experimental material.

6.4 Special replication requirements

All the above considerations support the common experimental setting: the evaluated algorithm processes a set of standardized circuits, and the resulting optimized circuits are measured. The stimulus – the input circuit – is the only thing that varies, and if the experiment covers an agreed-upon set of circuits, the results shall be valid and relevant to future performance of the algorithm.

Yet there are situations where this is not enough. The first case we describe is the situation where the algorithm has been deliberately randomized, that is, when some decision is taken randomly. Such randomness is a separate source of variance, and must be included into the design of the experiment.

The presence of variation can be unplanned, unexpected and hidden. Such variances are discovered only by chance; once known, they must be also accounted for. Fiser, Schmidt, and Balcárek (2014b) who document that such cases are common in academic and commercial software.

In both cases, the variance is additional. In order to achieve comparable confidence from statistics, much additional replication is needed.

6.4.1 Randomized algorithms

With randomized algorithms, some of the performance measures are random variables; the quality of results (in a broad sense) in the case of Monte Carlo algorithms, running time in the case
CHAPTER 6. EXPERIMENTAL EVALUATION

of Las Vegas algorithms.

The random nature of the algorithm is obvious, and the algorithm is understood as a *randomly behaving* object, as in Figure 6.8b. The algorithm is then measured and characterized in a probabilistic manner, which, of course, needs additional replication of each run of any experiment.

Figure 6.8: Two ways to formalize randomization

To interpret the observed data from an experiment, various assumptions are taken, which may not be always valid. The average value is commonly taken as the measure; a symmetric distribution is therefore assumed. Measurement errors and quantity variations in science have a Gaussian distribution so often that the mean and the standard deviation are used for comparison without stating the assumption.

For illustration purposes, we present the probability distribution of synthesized circuit size, for an externally randomized procedure (Fišer, Schmidt, and Balcárek 2014a). A closer analysis in

Figure 6.9: Solution quality of ABC *dch if lutpack* randomized by signals declaration, on the *cordic* circuit (Schmidt, Blažek, and Fišer 2014). The vertical line represents the best known solution.

Schmidt, Blažek, and Fišer (2014) shows that a feasible model is a sum of Gaussian distributions, similarly to the “bi-modal spread” in Puggelli et al. (2011).

Even if simple assumptions hold, the notion of “better” is not straightforward. Fair comparison becomes a complicated process requiring care (and huge computer time).

Shum and Anderson (2012) point out that many procedures of the ABC system are randomized, and only pretend deterministic behavior by fixing the seeds of the system’s pseudo-random number generator. They removed the fixation, and observed deviations in time and area up to 5%. Their view therefore corresponds to Figure 6.8.
6.4. SPECIAL REPLICATION REQUIREMENTS

6.4.2 Deterministic algorithms with hidden variance sources

In practice, most EDA procedures must behave deterministically, because designers need reproducibility. The designers also expect that they obtain similar results from similar inputs, and equivalent results from equivalent inputs.

The question is, what “equivalent” means in this context, and it is the designer who judges the equivalence. The input and output of an optimization procedure are equivalent in the sense that they implement the same Boolean function. The designer understands, however, that two circuits implementing the same functions are not equivalent by their area, speed and even optimization potential. Most designers, however, assume that any two structural descriptions of the same circuit are equivalent.

In a simulation language, the behavior of an FSM can be described in various ways, and the simulation results will always be identical. If such descriptions, however, are used for synthesis, the results will differ, some of them may be even unacceptable. Moreover, different synthesis tools will react differently to such inputs. This is, nevertheless, accepted by the design community, as it is thoroughly documented.

A designer writing a circuit description usually assumes that

- the exact form of any textual label (such as variable names or signal names) are semantically insignificant unless documented otherwise (e.g. naming conventions),
- the ordering of any list (such as the order of declarations or descriptions) is semantically insignificant unless documented otherwise, and
- equivalent control structures (unless documented otherwise), can be used interchangeably.

These assumptions are, surprisingly, violated by many EDA tools in some part.

Harlow III and Brglez (2001) evaluated heuristics that produce optimum variable ordering in ROBDDs. They included also the ‘null’ heuristics, which simply preserves the original ordering, into their experiments. It is natural to generalize this into the assumption that the original ordering may influence the resulting, optimized ordering. They discovered the following sources of variance:

- signal (variable) identifiers, and
- node ordering.

They do not discuss the variable declaration ordering in any way, nor do they bring any evidence that other sources of variation exist in the studied algorithms.

The experiments are designed to account for both discovered sources of variance. To achieve a fair cover of the additional variance sources, the authors introduced the methods of random identifier construction and random permutations. Probabilistic distribution of output measures (BDD size) is compared using Student t-test, which tells whether the results of two algorithms differ significantly or by chance. The algorithms themselves are considered to be deterministic (Figure 6.8a).

The experimental design is explained thoroughly, and contrasted to the “standard benchmarking procedure.” The same approach is taken in other related papers (Harlow III and Brglez 1998, Kapur, D. Ghosh, and Brglez 1997). The method is also generalized to use modified instances, which are not semantically equivalent (D. Ghosh et al. 1998, Brglez and Osborne 2007), however this is outside of our scope.

These texts seem to suggest that the authors aim to replace the limited number of available standardized circuits by a greater number of circuits derived from several “reference circuits.” Although such an approach limits computational resources needed, it invariably leads to an incomplete coverage (Figure 6.10b). We admit that some of the questions they have (e.g. the
CHAPTER 6. EXPERIMENTAL EVALUATION

(a) One class of equivalence from Harlow III and Brglez (2001)
(b) An incomplete cover as in Harlow III and Brglez (2001)
(c) A complete cover as in Fišer (2012)
(d) A complete cover with discovered discrete values as in Puggelli et al. (2011)

Figure 6.10: Coverage of additional variable sources in evaluation
dysfunctionality of one of the algorithms) can be answered using such stimuli. For evaluation in
general, we still need all the standard circuits, and to replicate each one by random names and
node permutation.

Multiple authors reported many EDA tools, both academic and commercial, to be sensitive
to the following sources of variance:

- signal identifiers (Puggelli et al. 2011),
- signal declaration (as opposed to signal/node description) order (Puggelli et al. 2011, Fišer
  and Schmidt 2011, Fišer, Schmidt, and Baláž 2014b)
- expression transformations (Puggelli et al. 2011),
- control structures transformations (Puggelli et al. 2011)

All the authors come to the conclusion that a single run on a set of circuits cannot be used for
comparison, as the variance of performance caused by these hidden sources of variation is greater
than those reported in papers presenting them. Quoting from Puggelli et al. (2011):

Since we have shown that synthesis results are spread across stochastic distribu-
tions, spurious improvements may be reported if the design lies at the extremes of the distributions.

In the cited papers, additional variation sources were again covered by random sampling of the particular domain, but the full range of circuits was retained (Figure 6.10c).

Fišer (2012), Fišer and Schmidt (2012c), and Fišer, Schmidt, and Balcařek (2014a) see such algorithms as randomized, presenting the idea of a designer who puts those details into the code at liberty, that is, randomly. From a designer’s point of view, such algorithms are nearly equivalent to true randomized algorithms; small changes in input can cause large differences in output, even if repeated runs on identical input produce the same output (as with randomized algorithms, because they pretend determinism by fixing their generators’ seeds). From the evaluation point of view, however, there is a different source of variance, namely permutations in the input description as opposed to the variance of the internal random number generator. Nevertheless, the presentation of the results is the same as with randomized algorithms: probabilistic distributions, not unique numbers. It can be therefore concluded that algorithms with hidden variance sources are, for most purposes, equivalent to randomized algorithms.

The two groups of algorithms requiring special replication in experiments presented above differ in one aspect: in randomized algorithm, the variance is a part of the algorithm itself, while in the other group it is embedded into the implementation (Figure 6.11).

![Figure 6.11: A summary of variance sources](image)

Experimental evaluation (as well as practical usage) always depends on a concrete implementation, and the two cannot be separated. Several factors increasing an implementation vulnerability were identified (Shum and Anderson 2012), (Fišer and Schmidt 2011), (Fišer, Schmidt, and Balcařek 2014b). All are related to container iteration.

**Incomplete specification.** If the algorithm specifies “for all nodes in topological order” (as it is common with AIG algorithms), it is an incomplete specification; there are multiple sequences of nodes which are in topological order.

**Transformation into data structures.** If the input description is transformed into an ordered BDD, then the variable ordering is likely to affect the size and form of the resulting BDD, which in turn can affect the performance of heuristics such as decomposition (C. Yang and Ciesielski 2002).

**List iteration.** Lists in the input description are often stored as arrays or linked lists, and are traversed in a lexicographical order. If that order has no (documented) meaning, the traversal introduces bias, which is affected by the (otherwise insignificant) input ordering.
Hash functions. A hash function transforms an object (such as a variable identifier) into a location of the object. When a collection of such object is iterated, the ordering depends on the objects themselves (such as the concrete identifiers).

Because these constructs can hardly be avoided in EDA software, the hidden sources of variance are likely to remain present.

6.5 ESPRESSO-MV evaluation

To illustrate the approaches outlined above, we suggest the ESPRESSO-MV memorandum (Rudell 1986) as an example of a carefully performed evaluation, and also as an example of the quantitative and qualitative aspects of the evaluation.

The Espresso family of algorithms, originally developed at IBM and continued at UC Berkeley, become a de-facto industrial standard for two-level logic minimization (Chapter 5). From the beginning, it included both heuristic (ESPRESSO) and exact minimization (ESPRESSO-exact). The algorithms evolved as ESPRESSO-IIAPL and ESPRESSO-IIC. The report describes a new version, ESPRESSO-MV, which can handle multi-valued functions.

6.5.1 Performance evaluation

The aim of the final experimental evaluation is to show that

- the algorithm provides solutions of acceptable quality, that is, the solutions for “industrial benchmarks” are reasonably close to optimum, yet the running time remains manageable;
- the algorithm performs better than its predecessors on “industrial benchmarks”.

Most of the benchmarks are identified as industrial designs. At that time, a two-level logic description was commonly implemented directly as a PLA structure in silicon, and therefore the original human input had the two-level form. As discussed later, this is a distinct design style and the designer may have adapted to it. A minority of the circuits realizes some mathematical function, and does not originate from industry.

As the first step, the author compared the exact minimization in ESPRESSO-MV with other existing minimizers. Besides the proofs of quantitative claims (elapsed time), the comparison had an important consequence, as the author put it:

*I am thus in an excellent position to grade the quality of the results for the heuristic minimization algorithm Espresso-MV. I know the minimum solution for 107 of the 134 examples in the test set, and, as shown in 6.2.3 I have a lower bound for 5 of the remaining 27 examples.*

For a minimization problem, any existing solution establishes an upper bound, and, of course, better solutions provide tighter upper bounds. Here, the exact minimization algorithm was adapted to provide also lower bounds for the instances it was unable to solve in the given time and memory.

6.5.2 Instance classification

Two-level minimization algorithms prior to ESPRESSO were often decomposed into two subsequent phases: prime implicant enumeration and solving the Unate Covering Problem (Coudert 1996). Although ESPRESSO does not require explicit formulation and solution of a covering problem
instance (which is one of the reasons the algorithm succeeded so well), input examples are classified by the covering problem instances they produce during solution. The classification is kept as a general measure of the examples. Besides the final demonstration, the characterization was probably used during the development, to focus on difficult examples and to study how the cyclic constraints affect the heuristics.

The classes are qualitative:

- trivial,
- non-cyclic,
- cyclic and solved,
- cyclic and unsolved,
- too many primes.

The actual classification is a by-product of the exact minimization performed earlier. “Too many primes” means that the covering problem instance could not be even constructed.

The classification of examples is most important in comparison with other exact minimizers. The author studies their performance on examples of different classes separately, and is therefore able to point out strong and weak points of the algorithms.

6.5.3 The origin of evaluation examples

Some of the examples, namely dk17 and dk27 resulted from FSM synthesis. The aim was to demonstrate how the algorithms perform on Boolean functions produced by a contemporary FSM minimization and encoding procedure.

In addition to the examples described above, a set of random examples used to demonstrate the PRESTOL-II minimizer (Bartholomeus and Man 1985) was added, because PRESTOL-II was not available for evaluation on other examples. They differed from practical circuits, and the differences could have been even characterized:

These random problems are especially difficult minimization problems because of the large percentage of don’t-care minterms, and the fact that the DC-set is scattered. As a result, all of these examples have a very large number of prime implicants, very few essential prime implicants, and most of them had cyclic constraints in the covering problem. Because these examples exhibit behavior much different from either the Industrial examples or the mathematical functions, these results have been presented apart from the rest of the test set.

6.5.4 Hidden sources of variance

According to (Fiser 2012, page 18), ESPRESSO is sensitive to the declaration order of both input variables and output variables. The average variation of output literal number is only slightly above 2%, but the maximum variation is 43%, well above any tolerable error. Similar sensitivity exists in the exact mode of ESPRESSO, which minimizes the number of terms, not literals. The sensitivity is measurable, but negligible even for evaluation purposes.

6.5.5 Summary of the ESPRESSO example

We can see many of the previously discussed features in this example. Both qualitative and quantitative measures of problem instances were used. The evaluation is based on practically relevant instances, with the LEKO or LEKU property. To achieve a straightforward comparison with other procedures, synthetic data were used, however, their characteristics proved to be
distinct from practical data. The output of preceding steps in design (FSM synthesis) was also used, even if that procedure was not mentioned or even documented.

### 6.6 Benchmark sets and support for evaluation

Experimental evaluation can become, as has been explained, a complicated process. Yet it cannot be substituted, and thus inaccurate or inadequate evaluation can do much harm. To give the process a sufficient support, we analyze the process of experimental evaluation (“benchmarking”) to a greater depth than usual.

#### 6.6.1 Benchmark sets

A benchmark is understood in all areas as “a reference point or reference item” (Merriam-Webster 2015, entry ‘benchmark’). So, the question is, what is the point of reference when evaluating an EDA procedure?

The answer is simple: everything what is needed to perform an experiment as established in natural science. It includes all environment, conditions, input, and measures. More importantly, it also includes the exact purpose of the experiment. Currently, a benchmark is only a part of such an experimental environment, namely the input. Every research paper that uses the input, in fact establishes a private benchmark in the full sense. Besides that, the knowledge about the benchmarks gets lost in seemingly unrelated sources.

#### 6.6.2 Benchmarking support

The prime requirement for experiments is their trustworthiness, leading to strict standards for repeatability. To repeat an experiment such as in Figure 6.4, the input data, together with procedure parameters, are in principle sufficient.

One minor problem is the accessibility of the formerly tested procedures. For this reason (among others), many academic programs are published including the source code. With a more complex setting, such as in Figure 6.5, the access to the surrounding transformations and measurements procedure is also required. This may be impossible or impractical. Instead, the intermediate data, that is, the immediate input to the evaluated procedure can be stored.

The most difficult situation is not with repeatability, but with interpretation. The scientific community can judge the presented conclusions as significant or relevant, or may wish to analyze them in greater depth. This is just another facet of trustworthiness: that the input is relevant to the problem. For such a continuation of the original work, full documentation is needed.

An example of full benchmarking environment is in Figure 6.12, where, for sake of simplicity, procedures are supposed to measure their input and/or output. To maintain the practical relevance of the experiment, the root of the graph shall always be a human input. If that is transformed, the transformation must also be fully documented (e.g. FSM encoding style in the example). If qualitative information about the data is known, it is represented by some of the attributes. Response metrics are documented together with measuring procedures.

Coverage of multiple additional variation sources, such as in Figure 6.10 require substantially greater number of input examples than the common few tens or few hundreds of circuits. To store all of them might not be practical; many of them, for example permuted descriptions of a single circuit, can be generated on request, and only parameters necessary for the generation must be stored. More techniques of this kind can be found to make such a repository feasible.

Such an experimental environment requires a considerable infrastructure and collaboration across the research community. It may seem an unneeded luxury and an overkill. We will
nevertheless document in Chapter 4 and subsequent chapters, that a loss of reliability already
has been effected by lost information and lacking benchmark data.

6.7 Summary

- Analytical evaluation is impractical and nearly impossible in the EDA field, partly because
  of algorithms complexity.
- Experimental algorithmics is a related field, but its focus on algorithms alone enables it to
  work with synthetic examples.
- Performance prediction and algorithms comparison are quantitative questions known from
  experimental algorithmics.
- Seeking understanding of an algorithm or procedure is a qualitative process.
- To find all sources of variance is still an open problem, even in experimental algorithmics.
- All known sources of variance shall be covered to provide the chosen degree of reproducibility.
- To maintain the practical significance of experiments in the EDA field, authentic examples
  must be used.
- Benchmark circuit sets must document the source of the circuit together with all manipulations
  with the description, and shall keep those manipulations to a minimum.
- Optimum or best known solutions shall be publicly kept for all problem instances where
  the circuit is a public benchmark circuit.
Chapter 7

Problems with synthesis of parity predictors

We detected the performance problems discussed here at the same time the paper by Cong and Minkovich (2007b) was published. Unlike that paper, we worked with real, not artificial circuits. We saw the phenomenon as an opportunity to construct small but difficult examples, and published a study (Fišer and Schmidt 2008) in that sense. As we show later, such examples would not be realistic.

7.1 The problem

Kubátová, Kubalík, and Fišer designed an architecture for reliable self-checking circuits (Kubalík, Fišer, and Kubátová 2006; Fišer, Kubalík, and Kubátová 2008) based on parity checking. The idea was, similarly to existing architectures, that two copies of a circuit will be employed. Instead of the usual output comparator, only parity (or several parities) will be compared. Thus one copy of the circuit becomes a mere parity predictor (Figure 7.1). Because less information is needed from the predictor, it was hoped that the predictor will be smaller than the full circuit, and the overhead of the architecture will be lessened. To prove this claim, experimental evaluation was necessary.

From their previous experience, they knew that synthesis tools tend to adhere to the circuit structure obtained from the designer. To be sure that such a structure was removed, they collapsed the whole circuit to SOP. The examples were constructed as follows:

1. Take any circuit with $m > 1$ outputs.
2. Construct a XOR tree with $m$ inputs.
3. Connect the tree to the outputs of the core circuit.
4. Collapse the entire circuit to obtain its two-level representation.

The circuit was then measured as outlined in Figure 7.2. The number of 4-input LUTs has been chosen as the circuit’s complexity metrics in correspondence with Cong and Minkovich (2007b). Because this metric is additive, an upper bound could be easily determined: the minimum number of 4-LUTs needed to construct the XOR tree is $\lceil (m - 1)/3 \rceil$, where $m$ is the number of the original circuit’s outputs. For example, the alu1 MCNC benchmark circuit having 8 outputs may be implemented by 8 4-LUTs. Then, the upper bound of the complexity of the 1-parity generator circuit would be $8 + 3 = 11$ 4-LUTs.
MCNC 3.0 benchmarks (S. Yang 1991a) were used for evaluation. Figure 7.3 summarizes the results. It plots the ratio of the actual size to the upper bound (which we call the *nastiness factor*) as a function of the circuit size, measured by its upper bound. We used three different synthesis processes (SUEs):

- **SIS** (Sentovich and al. 1992) under the script recommended for LUT synthesis
- **SIS** under the script *rugged* script, followed by `tech_decomp -a 4`, to obtain a network of 4-input nodes (i.e., LUTs as well)
- the LUT synthesis (*choice fpga lutpack*) in ABC (Berkeley Logic Synthesis and Verification Group 2000).

Each vertical line in the plot represents the span of size the above listed tools produced on a single MCNC benchmark circuit. If the nastiness factor is below 1, the circuit size is reduced when it is converted to a parity predictor, and the synthesis behaves as expected (the result size is below the upper bound). Circuits having the factor greater than 1 cause the synthesis to produce predictors larger than the upper bound.

Some “nasty” (upper part) and “nice” (lower part) examples are shown in Table 7.1. After the circuit name, the upper bound of the number of 4-LUTs is shown. Then the numbers of 4-LUTs obtained by the circuit collapsing and a subsequent synthesis by SIS (LUT synthesis script), SIS using a script *rugged* followed by `tech_decomp -a 4` and ABC is shown in the following columns. The respective nastiness factors are indicated in brackets. The benchmarks are sorted by their nastiness factors.

It can be seen that the “nastiest” benchmark is *alu*, with synthesis by SIS almost 26-times bigger than the upper bound. The second SIS optimization script *script_rugged* followed by
The tech_decomp -a 4 always yields worse results than the recommended one. This is a behavior similar to the LEKU examples by Cong and Minkovich, nevertheless the circuits are much smaller, which avoids objections raised by EDA vendors.

### 7.2 Robustness: it is not coincidence

It can be objected that the difficulty of the circuits is a result of random interaction between the core circuit and the XOR tree and that a small change in either of them will make the circuit ordinary. We altered the circuits and procedures in different ways and observed whether the circuits stop to make problems. Mostly, it was not the case.

#### 7.2.1 Robustness with respect to the number of parity trees

We tested the robustness of the problem first by splitting the outputs into groups and constructing a separate XOR tree for each group. In particular, we have varied the number of parity bits from 1 (which is the case of our examples) to \( m \) (which corresponds to an original benchmark, without any parity XORs – the core). We used two methods to produce the parity bits:

- A random assignment. The original circuit’s outputs are distributed among the XOR trees at random.
- A more sophisticated technique based on the analysis of the circuit (Fišer, Kubalík, and Kubátová [2008]).
7.2. ROBUSTNESS: IT IS NOT COINCIDENCE

Table 7.1: MCNC benchmarks ordered by nastiness, with LUT numbers as metrics

<table>
<thead>
<tr>
<th>Core circuit</th>
<th>Upper bound</th>
<th>SIS LUTs</th>
<th>SIS nastiness</th>
<th>SIS (2) LUTs</th>
<th>SIS (2) nastiness</th>
<th>ABC LUTs</th>
<th>ABC nastiness</th>
</tr>
</thead>
<tbody>
<tr>
<td>alu1</td>
<td>11</td>
<td>283</td>
<td>25.78</td>
<td>467</td>
<td>42.45</td>
<td>444</td>
<td>40.36</td>
</tr>
<tr>
<td>misc3c</td>
<td>258</td>
<td>1590</td>
<td>6.16</td>
<td>1753</td>
<td>6.79</td>
<td>2182</td>
<td>8.46</td>
</tr>
<tr>
<td>alu3</td>
<td>33</td>
<td>110</td>
<td>3.33</td>
<td>217</td>
<td>6.58</td>
<td>139</td>
<td>4.21</td>
</tr>
<tr>
<td>alu2</td>
<td>36</td>
<td>94</td>
<td>2.61</td>
<td>255</td>
<td>7.08</td>
<td>113</td>
<td>3.14</td>
</tr>
<tr>
<td>b12</td>
<td>32</td>
<td>68</td>
<td>2.13</td>
<td>123</td>
<td>3.84</td>
<td>81</td>
<td>2.53</td>
</tr>
<tr>
<td>t1</td>
<td>73</td>
<td>142</td>
<td>1.95</td>
<td>272</td>
<td>3.73</td>
<td>134</td>
<td>1.84</td>
</tr>
<tr>
<td>alu4</td>
<td>481</td>
<td>929</td>
<td>1.93</td>
<td>1330</td>
<td>2.77</td>
<td>1738</td>
<td>3.61</td>
</tr>
<tr>
<td>t4</td>
<td>17</td>
<td>27</td>
<td>1.59</td>
<td>82</td>
<td>4.82</td>
<td>28</td>
<td>1.65</td>
</tr>
<tr>
<td>mp2d</td>
<td>40</td>
<td>63</td>
<td>1.58</td>
<td>113</td>
<td>2.83</td>
<td>70</td>
<td>1.75</td>
</tr>
<tr>
<td>e64</td>
<td>744</td>
<td>23</td>
<td>0.03</td>
<td>39</td>
<td>0.05</td>
<td>22</td>
<td>0.03</td>
</tr>
<tr>
<td>prom1</td>
<td>2010</td>
<td>62</td>
<td>0.03</td>
<td>213</td>
<td>0.11</td>
<td>125</td>
<td>0.06</td>
</tr>
<tr>
<td>pope</td>
<td>188</td>
<td>7</td>
<td>0.04</td>
<td>23</td>
<td>0.12</td>
<td>14</td>
<td>0.07</td>
</tr>
<tr>
<td>bw</td>
<td>64</td>
<td>3</td>
<td>0.05</td>
<td>7</td>
<td>0.11</td>
<td>6</td>
<td>0.09</td>
</tr>
<tr>
<td>mainpla</td>
<td>2417</td>
<td>118</td>
<td>0.05</td>
<td>310</td>
<td>0.13</td>
<td>142</td>
<td>0.06</td>
</tr>
<tr>
<td>lin</td>
<td>294</td>
<td>15</td>
<td>0.05</td>
<td>72</td>
<td>0.24</td>
<td>30</td>
<td>0.10</td>
</tr>
<tr>
<td>al2</td>
<td>93</td>
<td>5</td>
<td>0.05</td>
<td>15</td>
<td>0.16</td>
<td>8</td>
<td>0.09</td>
</tr>
<tr>
<td>exps</td>
<td>518</td>
<td>28</td>
<td>0.05</td>
<td>74</td>
<td>0.14</td>
<td>36</td>
<td>0.70</td>
</tr>
<tr>
<td>ope</td>
<td>382</td>
<td>24</td>
<td>0.06</td>
<td>68</td>
<td>0.18</td>
<td>27</td>
<td>0.7</td>
</tr>
</tbody>
</table>

The results for four selected MCNC (S. Yang 1991a) benchmarks are shown in Figure 7.4. The numbers of parity bits are indicated on the x-axis (starting from 1 to the number of the circuit’s outputs), the y-axis shows the number of LUTs after the synthesis by SIS. The two curves correspond to the two grouping algorithms. Naturally the curves meet at their endpoints, where no optimization of the output grouping can be done. The upper curves correspond to the random grouping approach (yielding more LUTs). The values were obtained by averaging the results from 500 runs of the whole synthesis process with different variable ordering (different grouping). For all circuits tested, the sequences between endpoints are monotonous, similar to Figure 7.4. Benchmarks alu1 and alu2 are typical “nasty” circuits. Their size rapidly increases with decreasing the number of parity bits. On the contrary, 5xp1 and duke2 are the “nice” ones, where appending XORs to their outputs yields less logics, as expected.

7.2.2 Robustness with respect to different complexity metrics

In the previous measurement, we used the average number of LUTs as our metrics. Table 7.2 shows what happens when we change it to the average number of literals in a factored form, a radically different metric. The ordering is the same by both metrics. This parallel continues – although not precisely – for several more benchmarks, well into the region where the nastiness factor ceases to be interesting.

7.2.3 Robustness with respect to circuit change – alu1 in detail

As we stated before, problematic parity predictors are fairly small. This enables us to gain some insight by reverse engineering and structure alterations of the core circuit. alu1 is one of the smallest and yet nastiest circuits. Moreover, its name originally suggested the presence of iterative structures and handcrafted arithmetic circuitry. Many years later, we found out that
Figure 7.4: Synthesis results for variable number of XOR outputs, with random (upper) and intelligent (lower) output grouping

the circuit is the first stage of the famous 74181 ALU. Figure 7.5 is a detailed schematic of one possible implementation of the original PLA description. The core circuit is composed of four subcircuits, which all share four of the inputs (ALU operand selection). Each other input is processed by only one respective subcircuit. Three of the four subcircuits are identical, the fourth one is simplified (in the PLA form, one term is missing).

Figure 7.6 outlines our experimentation. First, we tried to determine if the irregularity (i.e., the missing term) has any significance. The regularized core circuit, \texttt{alu1RG}, showed almost identical complexity results when augmented with the XOR tree (Table 7.4). This encouraged us to derive a small circuit, \texttt{alu1HFRG}, with only subcircuits from Figure 7.5 i.e., a half of the \texttt{alu1RG} circuit. Even this toy example was not synthesized optimally. With circuits bigger
7.2. ROBUSTNESS: IT IS NOT COINCIDENCE

Table 7.2: MCNC benchmarks ordered by nastiness, with LUT number and factored form size as metrics

<table>
<thead>
<tr>
<th>name</th>
<th>nastiness by LUT#</th>
<th>nastiness by factored form</th>
</tr>
</thead>
<tbody>
<tr>
<td>alu1</td>
<td>25.78</td>
<td>8.12</td>
</tr>
<tr>
<td>misex3c</td>
<td>6.16</td>
<td>2.50</td>
</tr>
<tr>
<td>alu3</td>
<td>3.33</td>
<td>1.87</td>
</tr>
<tr>
<td>alu2</td>
<td>2.61</td>
<td>1.78</td>
</tr>
<tr>
<td>b12</td>
<td>2.13</td>
<td>1.03</td>
</tr>
<tr>
<td>t1</td>
<td>1.95</td>
<td>0.81</td>
</tr>
</tbody>
</table>

Figure 7.5: alu1 schematic

than alu1RG, we ran into difficulties with the synthesis tools. The collapsing processes of SIS and MVSIS either produced normal forms of enormous size (up to 357,000 terms in the case of alu1TWRG and MVSIS), or failed to collapse the circuit (alu1QDRG). It is problematic to distinguish fails caused by the intermediate form and by the nastiness of the circuit itself. When produced, the normal form size differed greatly between SIS and MVSIS, which usually does not occur with “nice” circuits and which had a significant impact on synthesis results. Therefore, the results in Table 7.4 are obtained by “best effort”.

Nevertheless, alu1TWRG (2-times alu1RG) had a record nastiness. The scatter points in Figure 7.3 seem to be contained within a convex body, which suggests that large nasty circuits may be sparse; alu1TWRG contradicts this.

Figure 7.6: alu1 derivatives
CHAPTER 7. PROBLEMS WITH SYNTHESIS OF PARITY PREDICTORS

Table 7.3: The nastiness of alu1 derivatives

<table>
<thead>
<tr>
<th>Name</th>
<th>Upper bound</th>
<th>SIS LUTs</th>
<th>SIS nastiness</th>
<th>ABC LUTs</th>
<th>ABC nastiness</th>
</tr>
</thead>
<tbody>
<tr>
<td>alu1</td>
<td>11</td>
<td>283</td>
<td>25.78</td>
<td>444</td>
<td>40.36</td>
</tr>
<tr>
<td>alu1RG</td>
<td>11</td>
<td>280</td>
<td>25.45</td>
<td>491</td>
<td>44.64</td>
</tr>
<tr>
<td>alu1HFRG</td>
<td>5</td>
<td>9</td>
<td>1.80</td>
<td>28</td>
<td>5.60</td>
</tr>
<tr>
<td>alu1TWRG</td>
<td>21</td>
<td>fail</td>
<td>7826</td>
<td>372.67</td>
<td>372.67</td>
</tr>
<tr>
<td>alu1QDRG</td>
<td>43</td>
<td>fail</td>
<td>fail</td>
<td></td>
<td></td>
</tr>
<tr>
<td>alu1ITWRG</td>
<td>21</td>
<td>fail</td>
<td>1088</td>
<td>51.81</td>
<td>51.81</td>
</tr>
</tbody>
</table>

Table 7.4: The nastiness of alu1 derivatives

To test the influence of the reconvergent fan-out, we cut the four common inputs of alu1TWRG and gave each half of the circuit a separate set of four inputs in alu1ITWRG. Although the normal form was bigger than that of alu1TWRG, the nastiness dropped somewhat. It is still to be determined whether this is significant.

7.3 Symmetry: it is not a cause

The paper F. Wang and Dietmeyer (1998), although aimed at synthesis improvement, brings observations that are probably relevant to our circuits. They construct an evaluation circuit by a method close to ours. They compose a MCNC benchmark, namely $f51nm$, with the symmetrical circuit $S^{(4)}$. (In this notation, the XOR tree would be $S^{(1,3,5,7)}$). The purpose of this circuit is to demonstrate the advantages of a synthesis process that can handle near symmetry in circuits well. They also state that common algebraic decomposition methods perform poorly not only on symmetric functions, but also on functions exhibiting near symmetry; moreover, that symmetry is not handled well in two-level minimization. These assertions, although intuitive, are not supported in the paper. Nevertheless, they give a possible explanation of the phenomena observed:

- Symmetric functions have logarithmic depth (and hence, linear circuit size) (Brodal and Husfeldt 1996).
- They are not handled well by contemporary algorithms.
- Near symmetry is in these respects similar to symmetry.
- Therefore, circuits with near symmetry tend to have small implementations, which are hard to find by contemporary algorithms.

As a check, we measured the first-order symmetry of the predictors using the ABC command print_symm. The results are summarized in Table 7.5. Typically, one third of all input pairs exhibit symmetric behavior, and this is unrelated to the fact whether the circuit is “nasty” (the upper half of the table) or “nice” (the lower half).

Symmetric functions, because of their efficient but hard-to-find implementations, have been the subject of intensive research for a long time (McCluskey 1956, Klir and Seidl 1966, Section IV). Yet from this measurement, we can conclude that symmetry is not the cause of performance problems.
7.4 XOR decomposition and global approach: requirements

When inspected manually, the resulting circuits from the investigated processes did not show any remnants of the output XOR tree. That suggested using a procedure which is designed to discover and produce XOR elements, namely the BDD-Based Logic Optimization System (BDS) (C. Yang and Ciesielski 2002).

To study the influence of the classical style synthesis and XOR decomposition, we prepared the data flows in Figure 7.7. The initial SOP form was obtained by the ABC command collapse. Then, the circuits were processed by BDS. We forced the program to use a global BDD for the entire input (BDS -useglb), as the default settings trade quality for memory consumption too eagerly. In an alternative run, we disabled the XOR decomposition completely (BDS -useglb -xhc).

A mapping procedure produces the desired metrics (number of LUTs). To mimic real design flow, we used the ABC command sequence dch, if, lutpack. The sequence optimizes the circuit first by resynthesis, then resynthesizes the circuit again to fit into 4-LUTs. This kind of resynthesis can be iterated, and the authors of ABC recommend to do so (Berkeley Logic Synthesis and Verification Group 2000). To measure the influence, we mapped the decomposed circuit either with one or with twenty iterations of the entire command sequence.

The results are in Table 7.6. To keep it readable, we present only sizes relative to the upper bound from Table 7.1. For comparison, the values form that table are repeated first. Columns with relative sizes for BDS without the XOR decomposition follow, with simple or iterated mapping and optimization. The results obtained from full BDS are in the final columns.

The following phenomena can be observed in Table 7.6 and Figure 7.8.

---

1The -xhc option is a local extension to BDS, and corresponds to the -xhardcore option in BDS-PGA
Table 7.6: MCNC benchmarks ordered by original nastiness, BDS-based synthesis, sizes relative to upper bound

<table>
<thead>
<tr>
<th>Core circuit</th>
<th>SIS</th>
<th>SIS(2)</th>
<th>ABC</th>
<th>BDS-xhc simple</th>
<th>BDS-xhc iter.</th>
<th>BDS simple</th>
<th>BDS iter.</th>
</tr>
</thead>
<tbody>
<tr>
<td>alu1</td>
<td>25.73</td>
<td>42.45</td>
<td>40.36</td>
<td>11.45</td>
<td>11.36</td>
<td>1.36</td>
<td>1.36</td>
</tr>
<tr>
<td>misex3c</td>
<td>6.16</td>
<td>6.79</td>
<td>8.46</td>
<td>1.43</td>
<td>1.43</td>
<td>0.85</td>
<td>0.83</td>
</tr>
<tr>
<td>alu3</td>
<td>3.33</td>
<td>6.58</td>
<td>4.21</td>
<td>1.55</td>
<td>1.58</td>
<td>1.03</td>
<td>1.03</td>
</tr>
<tr>
<td>alu2</td>
<td>2.61</td>
<td>7.08</td>
<td>3.14</td>
<td>2.36</td>
<td>2</td>
<td>2.58</td>
<td>2.5</td>
</tr>
<tr>
<td>b12</td>
<td>2.13</td>
<td>3.84</td>
<td>2.53</td>
<td>0.72</td>
<td>0.75</td>
<td>0.5</td>
<td>0.53</td>
</tr>
<tr>
<td>t1</td>
<td>1.95</td>
<td>3.73</td>
<td>1.84</td>
<td>0.77</td>
<td>0.77</td>
<td>0.62</td>
<td>0.6</td>
</tr>
<tr>
<td>alu4</td>
<td>1.93</td>
<td>2.77</td>
<td>3.61</td>
<td>1.46</td>
<td>1.39</td>
<td>1.02</td>
<td>0.92</td>
</tr>
<tr>
<td>t4</td>
<td>1.59</td>
<td>4.82</td>
<td>1.65</td>
<td>0.71</td>
<td>0.71</td>
<td>0.76</td>
<td>0.71</td>
</tr>
<tr>
<td>mp2d</td>
<td>1.58</td>
<td>2.83</td>
<td>1.75</td>
<td>0.75</td>
<td>0.75</td>
<td>0.38</td>
<td>0.4</td>
</tr>
<tr>
<td>e64</td>
<td>0.03</td>
<td>0.05</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>prom1</td>
<td>0.03</td>
<td>0.11</td>
<td>0.06</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>pope</td>
<td>0.04</td>
<td>0.12</td>
<td>0.07</td>
<td>0.04</td>
<td>0.04</td>
<td>0.06</td>
<td>0.05</td>
</tr>
<tr>
<td>bw</td>
<td>0.05</td>
<td>0.11</td>
<td>0.09</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>mainpla</td>
<td>0.05</td>
<td>0.13</td>
<td>0.06</td>
<td>0.05</td>
<td>0.04</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td>lin</td>
<td>0.05</td>
<td>0.24</td>
<td>0.1</td>
<td>0.07</td>
<td>0.06</td>
<td>0.06</td>
<td>0.05</td>
</tr>
<tr>
<td>al2</td>
<td>0.05</td>
<td>0.16</td>
<td>0.09</td>
<td>0.06</td>
<td>0.05</td>
<td>0.06</td>
<td>0.05</td>
</tr>
<tr>
<td>exps</td>
<td>0.05</td>
<td>0.14</td>
<td>0.07</td>
<td>0.05</td>
<td>0.04</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>opa</td>
<td>0.06</td>
<td>0.18</td>
<td>0.07</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
</tr>
</tbody>
</table>

- The pathologically bad results disappeared almost completely when using full BDS.
good results did not change much. With the exception of \textit{alu1} and \textit{alu2}, the result size is below or close to the upper bound.

- Iterated optimization does help, but is not the principal contributor to result quality. Therefore, the performance depends strongly on BDS.

- BDS in global BDD mode understands any input as behavior, and hence does not suffer from structural bias (Section 5.3). This property alone, even without the XOR decomposition, improves the result quality. The biggest gain is in the originally difficult cases.

- XOR decomposition improves the quality further. In the most difficult cases, the influence is even greater than what the global approach gains. The combined gain is up to one order of magnitude, but diminishes in the easy cases.

Two causes of the originally poor performance can be identified: structural bias and missing XOR decomposition, or an analogous ability to produce XOR circuits. They mostly affect the “nasty” cases. In the “nice” cases, most of the output XOR tree became absorbed into the circuit. First, the parity predictor is then no longer a XOR-intensive circuit, and the ability to produce XOR circuits is not important. Second, the circuits are rather small (as hoped for), which helps the procedures to overcome structural bias. When the XOR tree could not be absorbed, both the inability to overcome structural bias of the input SOP (more a trap than bias) and to synthesize XOR circuits is essential.

Since the original study, we succeed to obtain new solutions, in many cases substantially improved. We used massively iterated synthesis tools, externally randomized to maintain diversity (Fišer, Schmidt, and Balcář 2014a). Also, CGP (Fišer, Schmidt, Vašíček, et al. 2010) was used as a part of the optimization chain.

When the sizes of the circuits are taken relatively to the new results instead of the previous upper bound, the picture changes dramatically (Table 7.7 and Figure 7.9). The worst cases (\textit{alu1}, \textit{misc23c}) remain worst cases still, and the contribution of both structural bias resistance and XOR decomposition is still visible. While in Figure 7.8 the two classes of “nastiest” and
Table 7.7: MCNC benchmarks ordered by original nastiness, BDS-based synthesis, sizes relative to known minima; SIS, SIS(2) and ABC are the three procedures described in Section 7.1; procedures using BDS as described in Section 7.4 and Figure 7.7.

<table>
<thead>
<tr>
<th>Core circuit</th>
<th>SIS</th>
<th>SIS(2)</th>
<th>ABC</th>
<th>BDS -xhc</th>
<th>BDS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>simple</td>
<td>iter.</td>
<td>simple</td>
<td>iter.</td>
<td></td>
</tr>
<tr>
<td>alu1</td>
<td>28.3</td>
<td>46.7</td>
<td>44.4</td>
<td>12.6</td>
<td>125</td>
</tr>
<tr>
<td>misex3c</td>
<td>66.25</td>
<td>73.04</td>
<td>90.92</td>
<td>15.42</td>
<td>370</td>
</tr>
<tr>
<td>alu3</td>
<td>4.58</td>
<td>9.04</td>
<td>5.79</td>
<td>2.13</td>
<td>52</td>
</tr>
<tr>
<td>alu2</td>
<td>2.61</td>
<td>7.08</td>
<td>3.14</td>
<td>2.36</td>
<td>72</td>
</tr>
<tr>
<td>b12</td>
<td>7.56</td>
<td>13.67</td>
<td>9</td>
<td>2.56</td>
<td>24</td>
</tr>
<tr>
<td>t1</td>
<td>6.76</td>
<td>12.95</td>
<td>6.38</td>
<td>2.67</td>
<td>56</td>
</tr>
<tr>
<td>alu4</td>
<td>3.16</td>
<td>4.52</td>
<td>5.91</td>
<td>2.39</td>
<td>668</td>
</tr>
<tr>
<td>t4</td>
<td>2.7</td>
<td>8.2</td>
<td>2.8</td>
<td>1.2</td>
<td>12</td>
</tr>
<tr>
<td>mp2d</td>
<td>3.94</td>
<td>7.06</td>
<td>4.38</td>
<td>1.88</td>
<td>30</td>
</tr>
<tr>
<td>e64</td>
<td>1.35</td>
<td>2.29</td>
<td>1.29</td>
<td>1.29</td>
<td>24</td>
</tr>
<tr>
<td>prom1</td>
<td>1.68</td>
<td>5.76</td>
<td>3.38</td>
<td>1.76</td>
<td>65</td>
</tr>
<tr>
<td>pope</td>
<td>1</td>
<td>3.29</td>
<td>2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>bw</td>
<td>1.5</td>
<td>3.5</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>mainpla</td>
<td>5.13</td>
<td>13.48</td>
<td>6.17</td>
<td>5.52</td>
<td>105</td>
</tr>
<tr>
<td>lin</td>
<td>1.88</td>
<td>9</td>
<td>3.75</td>
<td>2.5</td>
<td>18</td>
</tr>
<tr>
<td>al2</td>
<td>1.25</td>
<td>3.75</td>
<td>2</td>
<td>1.5</td>
<td>5</td>
</tr>
<tr>
<td>exps</td>
<td>1.87</td>
<td>4.93</td>
<td>2.4</td>
<td>1.67</td>
<td>23</td>
</tr>
<tr>
<td>opa</td>
<td>1.5</td>
<td>4.25</td>
<td>1.69</td>
<td>1.31</td>
<td>21</td>
</tr>
</tbody>
</table>

“nicest” cases were clearly distinguishable, here it is not so. In the light of the better minima, the performance is not connected to the original nastiness. Some examples (misex3c, originally a “nasty” circuit, and mainpla, which is “nice”) have been still synthesized inefficiently.

The comparison with SIS is also interesting. The procedures using SIS contain decomposition as well. Although BDS uses better algorithms based on decision trees, the only significant difference is the XOR decomposition. Thus it appears as a significant requisite to a system synthesizing parity predictors.

7.5 Summary

- Many parity predictors are difficult to synthesize in the SOP form for most of synthesis systems.
- The difficulty (“nastiness”) can be measured using the upper bound obtained by separate synthesis of core circuits and parity trees.
- The nastiness is robust with respect to the choice of size-related metrics.
- The nastiness is robust with respect to the number of parity trees and other structure-preserving alterations of the circuit.
- Replication (even partial replication) of nasty circuits can lead to bigger nasty circuits. Reconvergent fan-out can make the circuits harder.
- XOR decomposition is necessary to obtain acceptable solutions.
- For any flow that includes a global SOP construction, either the classical synthesis approach must be used, or extreme resistance to structural bias is needed.
- The adverse influence of tech_decomp is probably not specific to parity predictors.
• The cause of poor results from BDS synthesis is also probably not related to parity predictors.
Chapter 8

Problems with the G25 examples

In this case, we did not focus on the problematic circuit example as in the previous chapter, but on other circuits exhibiting a similar behavior. We found out that the artificial nature of the original example is not the culprit. The example, however, has not been synthesized in its original state, but transformed. Hence, we followed the experimental procedure designed in Section 6.2.3 (Figure 6.6).

8.1 The problem

Cong and Minkovich (2007b) published examples targeted at the mapping phase of logic synthesis – the LEKO examples, and proved the optimum mapping for them. These examples have description variants, which can be used to test the entire logic synthesis. Only the upper bound of the optimal implementation size is known for synthesis (LEKU examples), they have a two-level unbalanced (LEKU-CD) structure and proved to be very hard for any synthesis process. Typical resulting circuits are more than 400-times larger than expected. There are also balanced versions (LEKU-CB), which are less hard (up to 5-times larger results).

The authors proved that structural parameters of the circuits are similar to practical circuits. To our knowledge, the cause of the observed (and rather alarming) synthesis results is not known, nor is the linkage between them and practical performance of synthesis tools.

The set of examples is defined using

- a relatively small circuit described as a Boolean network with two-input nodes,
- a replication algorithm which can produce circuits of unlimited size, ensuring that there is a path from each input to each output,
- and a proof of optimum mapping for the replicated circuit.

When used for synthesis evaluation, the circuit synthesized from an altered description is compared with the proved optimum mapping. As the synthesis can produce a better result than mere mapping, the optimum mapping is only an upper bound of the result size, and hence the name – Logic synthesis Examples with Known Upper bounds (LEKU).

To produce the altered synthesis input, the original description is collapsed into a SOP which is then converted into a structural description by applying the SIS command tech_decomp (Sentovitch and al., 1992) or the ABC command balance (Berkeley Logic Synthesis and Verification Group, 2000) (Figure 8.1). A network of two-input gates is obtained as a result.

The circuits are primarily meant as mapping examples. In a “textbook” interpretation, mapping operates on a structure (represented by a graph) and produces a structure composed of
look-up tables. Such processes are frequently based on FlowMap (Cong and Ding [1994]) or bin packing (Francis, Rose, and Vranesic [1991]) algorithms.

In the original experiments, however, commercial tools were used, where the mapping phase cannot be separated from other logic synthesis steps. Moreover, even ABC and SIS are built this way to a degree; their mapping commands or recommended mapping scripts include substantial amount of resynthesis. Therefore, we understand the examples as general logic synthesis examples.

LEKU-CD and LEKU-CB examples based on the G25 circuit were used, as they suffice to demonstrate the quality of the tested tools. The G25 circuit is known to have an upper bound of 70 4-LUTs. The CB version of the G25 circuit (CB(G25)) has 814 two-input gates, while CD(G25) is substantially larger, having 1167054 two-input gates.

8.2 Projected impact and Cong’s conjecture

The paper by Cong and Minkovich was written with several impacts in mind:

- “There may be something rotten in the state of logic synthesis”; the examples may lead to (even substantial) improvement.
- As synthesis tools improve, less competent designers are able to use them; tools shall be resistant to all sorts of inefficient code these designers write.
- Redundancy of a particular sort may be introduced to examples to test how resistant the tools are against that redundancy.

In a discussion of the examples, probably republished as (ICEach.com 2007), Cong is quoted:
“In theory, if you have a good synthesis tool, you should be able to rediscover the original circuit,” Cong said, “but all the methods we tried can not find the original structure, which is the reason we get the results 70 times larger.”

In other words, structural bias (Section 5.3) is responsible for the results. The authors, however, tested tools that share the resynthesis approach (Section 5.8). Using tools that do not depend on the structure of the input description, this conjecture can be tested.

8.3 G25 LEKU examples and diverse synthesis procedures

To obtain orientation in the problem, we synthesized the examples by several synthesis procedures, representing both the classical approach and resynthesis.

8.3.1 Circuits, transformations and metrics

In all cases, representations of the G25 circuit were used. They have been transformed from the original circuit by collapsing (ABC collapse), balancing (ABC balance), and decomposition into two-input gates (SIS tech_decomp) (Figure 8.2).

We found the role of tech_decomp critical. It moves, in effect, information from the behavioral domain (node functions in a Boolean network) into the structural domain, preventing all tools from producing an optimized internal representation of the circuit.

The principal property of the result we were interested in was whether a particular result is acceptable or shall be considered unacceptable (for example, multiple orders of magnitude larger than necessary), as presented, in Cong and Minkovich (2007b). For such an observation, almost any circuit measure suffices, and it is not very important whether the tool optimizes for speed or area.

The size of the resulting circuit was therefore measured as the number of 4-input LUTs, similarly to Mishchenko, Chatterjee, and Brayton (2006) and for similar reasons, namely to obtain a unified measure for decomposition algorithms working with different circuit representations.

8.3.2 Synthesis procedures

We used the following tools for decomposition: SIS (Sentovitch and al. 1992), ABC (Berkeley Logic Synthesis and Verification Group 2000), BDS (C. Yang and Ciesielski 2002), and BiDecomp (Mishchenko, Steinbach, and Perkowski 2001). Mapping by SIS was performed by running the script proposed in Sentovitch and al. (1992) for LUT synthesis, modified for mapping into 4-LUTs. ABC synthesis was performed by running the resyn2 script. The ABC dsd command (Bertacco and Damiani 1997) was also used to perform disjoint-support decomposition. Mapping in ABC was done by the fpga command, or by repeatedly running the command sequence strash, balance, fpga. In the case of BDS running on some examples, we had to apply a heuristic measure preventing the program from generating an infinite decomposition tree. We estimate that performance up to 10% better could be achieved with ideal cycle prevention.

8.3.3 Results

The first series of experiments was to merely synthesize the unaltered G25 circuit by SIS, ABC, and BDS with ABC mapping in the above described configurations. The size of the circuits obtained varied from 72 (ABC with resynthesis) to 136 (SIS) LUTs. BDS decomposition did not improve the result.
The second series of experiments aimed at the reproduction of the results presented in Cong and Minkovich (2007b). Prohibitively large circuits were obtained for the $CD(G25)$ circuit. Repeated application of the FPGA mapping command sequence did not yield results better than roughly 37,000 LUTs (more than 520-times the optimum). Resynthesis did not significantly help as well. We were unable to process $CD(G25)$ by BDS and BiDecomp, since both applications crashed for this example. The $CB(G25)$ circuit was problematic for all synthesis processes, however, the obtained results were no more than 2.7-times the optimum, thus acceptable.

The third series of experiments was performed on $G25$ in a SOP form. Various tools were used to obtain the SOP in the PLA format: ABC, MVSIS, SIS. The SOP obtained by ABC had roughly 19,000 terms and 14,000 terms after minimization with ESPRESSO. The other collapsing tools gave SOPs roughly ten times larger than ABC. All these SOPs were synthesized by ABC and the combination of BDS and ABC in the above described configurations. In some cases, BiDecomp with ABC mapping was also tested. In all cases, both BDS with ABC and BiDecomp with ABC were better than ABC alone.

The fourth series also used SOP forms of the circuits, in this case the SOPs of $CB(G25)$ and $CD(G25)$. Again, all the SOPs were synthesized as in the previous series.

The results from selected experiments are presented in Table 8.1. It can be seen that ABC managed to synthesize $G25$ collapsed by ABC and MVSIS quite well, synthesis of $G25$ collapsed by SIS yielded much worse results (900 LUTs after resynthesis). However, when BDS was run prior to the ABC LUT mapping, equal results of 113 LUTs were obtained in all cases. Even better results were obtained by running the ABC command dsd prior to the LUT mapping.

Surprising results were obtained from pre-processing the collapsed $G25$ minimized by Espresso. Excessively large results were obtained by ABC (1520 LUTs after resynthesis). However, BDS and ABC dsd, again, were able to rediscover the circuit structure. Espresso was unable to minimize the circuits collapsed by MVSIS and SIS, hence those results are not presented.

Notice, that when the $CB(G25)$ circuit was collapsed by ABC, the resulting PLA was equal to the PLA of the collapsed original G25, and thus the synthesis results were equal as well. Very similar results were obtained by collapsing $CB(G25)$ by MVSIS and SIS, hence the results are not presented.

Similarly, when $CD(G25)$ was collapsed by ABC, a PLA of a very similar size to the collapsed original $G25$ was produced. At the end, synthesis results were equal to the results obtained from the collapsed $G25$. The ABC dsd command generated the same result as in all the previous cases. We did not succeed in collapsing $CD(G25)$ by MVSIS or SIS.

Figure 8.2 summarizes the data flow and main results from the experiments. Where multiple results were obtained using different tools and/or different versions of input data, a range is indicated in Figure 8.2. Notice that the number of literals generally follows the number of terms. The only exception is $G25$ minimized by ESPRESSO, which decreased the number of terms but increased the number of literals.

### 8.3.4 Discussion

If $G25$ was a practical circuit, and if it was described in the original form, then there would have been no problems with synthesis. The first series of experiments shows that common synthesis tools can optimize the original circuit with acceptable results.

To alter the description into a SOP also does not cause problems of the scale discussed here, even if the results are not good.

For the problems to emerge, the description must be collapsed and processed by tech.decomp. In other words:

- the existing structure of the circuit must be discarded, and
• the resulting description must be understood as a structure, not a behavior.

Tools that treat input as behavior succeed. Two scenarios are possible:

• use the classical approach (Section 5.6), which builds a SOP and treats it as behavior, or
• build a global BDD first, as in ABC dsd.

From these observations, we conclude that the problems are indeed caused by structural bias. Thus, Cong’s conjecture can be supported, but only for tools having structural bias, such as those based on the resynthesis approach.

It remains a possibility, however, that such a behavior is triggered by some property of G25, which is not a practical circuit, and is otherwise irrelevant.

8.4 Synthesis of benchmark circuits in decomposed SOP form

To find out practical relevance, we repeated the measurements with benchmark circuits. The aim of the experiments was to determine how the synthesis result size depends on the size of input.
### 8.4. BENCHMARK CIRCUITS IN SOP FORM

Table 8.1: Synthesis results obtained for altered forms of circuits

<table>
<thead>
<tr>
<th>Input</th>
<th>SOP size [terms]</th>
<th>Tools combination</th>
<th>Result size [LUTs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>G25</td>
<td>–</td>
<td>ABC fpga, ABC resyn2 + fpga, ABC dsd + fpga, SIS</td>
<td>80, 72, 102, 136</td>
</tr>
<tr>
<td></td>
<td>18905</td>
<td>ABC fpga, ABC resyn2 + fpga, ABC dsd + fpga, BDS + ABC fpga, BiDecomp + ABC fpga, SIS</td>
<td>280, 200, 102, 113, 188, failed</td>
</tr>
<tr>
<td></td>
<td>218116</td>
<td>ABC fpga, ABC resyn2 + fpga, ABC dsd + fpga, BDS + ABC fpga, SIS</td>
<td>232, 170, 115, 113, failed</td>
</tr>
<tr>
<td></td>
<td>121500</td>
<td>ABC fpga, ABC resyn2 + fpga, ABC dsd + fpga, BDS + ABC fpga, SIS</td>
<td>1140, 900, 102, 113, failed</td>
</tr>
<tr>
<td></td>
<td>1755530</td>
<td>ABC fpga, ABC resyn2 + fpga, ABC dsd + fpga, BDS + ABC fpga, SIS</td>
<td>1725, 1520, 102, 113, failed</td>
</tr>
<tr>
<td></td>
<td>201816</td>
<td>ABC fpga, ABC resyn2 + fpga, ABC dsd + fpga, BDS + ABC fpga, SIS</td>
<td>14619, 245, 180, 424, failed</td>
</tr>
<tr>
<td></td>
<td>248967</td>
<td>ABC fpga, ABC resyn2 + fpga, ABC dsd + fpga, BDS + ABC fpga, SIS</td>
<td>18509, 218116, 280, 200, 102, 113, 188, failed</td>
</tr>
<tr>
<td>CB(G25)</td>
<td>–</td>
<td>ABC fpga, ABC resyn2 + fpga, ABC dsd + fpga, BDS + ABC fpga, BiDecomp + ABC fpga, SIS</td>
<td>286, 189, 102, 245, 180, 424, failed</td>
</tr>
<tr>
<td>CB(G25)</td>
<td>18905</td>
<td>ABC fpga, ABC resyn2 + fpga, ABC dsd + fpga, BDS + ABC fpga, BiDecomp + ABC fpga, SIS</td>
<td>280, 200, 102, 113, 188, failed</td>
</tr>
<tr>
<td>CD(G25)</td>
<td>–</td>
<td>ABC fpga, ABC resyn2 + fpga, ABC dsd + fpga, BDS, BiDecomp, SIS</td>
<td>40k, 27k, 102, failed, failed, failed</td>
</tr>
<tr>
<td>CD(G25)</td>
<td>18509</td>
<td>ABC fpga, ABC resyn2 + fpga, ABC dsd + fpga, BDS + ABC fpga, BiDecomp + ABC fpga, SIS</td>
<td>280, 200, 102, 113, 188, failed</td>
</tr>
</tbody>
</table>
8.4.1 Circuits, transformations and metrics

490 circuits were measured, from the following sources:

- LGSynth’93 benchmark set (McElvain 1993), 129 circuits; at the time, we were not aware of the manipulation of the set (Section 4.3).
- MCNC benchmark set (S. Yang 1991a), 103 circuits
- Altera OpenCores, converted to BLIF by Alan Mishchenko using Quartus II with standard settings, 110 circuits; may contain adders (Section 5.4)
- ISCAS’85 benchmark set (Brglez and Fujiwara 1985), 11 circuits
- ISCAS’89 benchmark set (Brglez, Bryan, and Kozminski 1989), 50 circuits
- LGSynth’91 benchmark set (S. Yang 1991b), 2 circuits
- Illinois set, 7 circuits (Chickermane, Lee, and Patel 1992)
- LEKU examples by Cong and Minkovich, 8 circuits
- ITC’99 benchmark set (Corno, Reorda, and Squillero 2000), 22 circuits
- manually encoded $n$-bit adders, by Petr Fišer, 16 circuits

The experiment comprised a reference branch, where the measured synthesis procedure has been run on an unaltered input, and an evaluation branch, where the same procedure processed an altered (transformed) input. The ratio of both input sizes is termed the input blow-up, and similarly the ratio of sizes resulting from the measured procedure is called the output blow-up (Figure 8.3). All input sizes were measured in literals, and all output sizes in 4-input LUTs for reasons discussed above in Section 8.3.1.

Three methods were used to produce SOPs of varying size. The first one was the SIS collapse command, which, when successful, produced large SOPs. The ABC collapse command produced smaller SOPs. We were able to randomize this command by declaration order (Fišer 2012), and get more different SOPs. The last method used BDDs to collapse a circuit by traversing all paths from root to the “1” terminal (Rucký 2007). As the shape and size of a BDD strongly depends on the order of variables (Bollig and Wegener 1996; Nagayama et al. 2003), the method can be effectively randomized by forced variable order. This method produces descriptions in DSOP form. Both randomized procedures were run 50 times. A blow-up in the order of 100 was obtained on more than ten circuits. Some of them were arithmetic circuits (dalu, fractional multipliers s420, s838, s839), some were not (ITC_b12). In accordance with circuit and communication complexity, high SOP blow-up is to be expected in multipliers.

The obtained data sets were examined for representativeness by several rules of thumb. Data for a circuit were discarded if

- they did not have SOPs with at least five different sizes, or
- their size did not cover at least a span of 1:1.6, or
- there was a gap in the sizes larger than approximately the third root of the span.

From the total 490 circuits, we managed to collapse 334, and from them, 143 were selected using the above criteria.

8.4.2 Synthesis procedures

Four variants of ABC synthesis were measured. All of them were based on the dch synthesis command, followed by if technology mapping (Section 5.8.3). lutpack is a post-mapping optimization procedure (Mishchenko, Brayton, and Chatterjee 2008). Alternatively, nfs (Mishchenko, Brayton, J.-H. R. Jiang, et al. 2011) was used as the last step instead, since this command is able to utilize flexibility in Boolean networks.
8.4. BENCHMARK CIRCUITS IN SOP FORM

Figure 8.3: Data flow of the SOP synthesis experiment

The additional layer of iteration suggested in Fišer and Schmidt (2012b) and Fišer (2012) and discussed in Section 5.8.4 was also tried, in the form of 20 iterations of the sequences dch if lutpack and dch if mfs described above.

These procedures have known sources of variance, which would introduce noise into the measurement. To cover at least the most prominent one, each procedure was run 50 times with permuted input, and the output size was averaged over the permutations.

8.4.3 Fišer’s conjecture

When quickly overlooking the first data, Petr Fišer noticed that the result size is proportional to the input size. Such a behavior agrees with the observation of Cong and Minkovich; one has only to produce a large enough SOP to obtain poor results from a synthesis procedure.

On closer examination, this assertion appeared not universal. Synthesis behaved as it should have on certain circuits, producing always nearly the same result. Sometimes, the proportionality was evident. Therefore, we evaluated the data with respect to Fišer’s conjecture, and created
models of typical behavior of the synthesis procedures.

### 8.4.4 Models

We assumed the following models for input blow-up – output blow-up dependency. To compare, we could not use confidence intervals, because we cannot make reliable assumptions about statistical properties of the input. Instead, we compared the models on a circuit-by-circuit basis using the Residual Sum of Squares (RSS).

**Constant regression**

Modeling the output blow-up with a constant corresponds to the ideal optimization performance. A difference (intercept) may exist, as the reference branch in Figure 8.3 works with the original structure while the evaluation branch always takes a SOP.

**Linear regression**

With unity slope, linear regression models a procedure that does not optimize at all. With the slope less than one, the procedure improves the result by a constant ratio. An intercept may appear for the same reason as in the previous case. Owing to the relative nature of the variables (relative output blow-up, not LUTs), we need not worry about the relationship between literals and LUTs. Moreover, linear regression contains the constant regression as a special case.

**Linear regression in the logarithmic domain**

Originally, the dependency has been observed in log-log scatter plots. While zero-intercept linear regression does appear as a line in a log-log plot, it is not so in the general case. Therefore, we considered a model which is a linear regression in a log-log plot.

**Nonlinear regression – the $rc$ function**

Ideal optimization and no optimization can be seen as two extreme cases of behavior. We sought a model that would asymptotically behave as these two, but would unify them into a single model. We found inspiration in frequency response of simple RC analog circuits. A low pass RC circuit has little attenuation up to, say, half the critical frequency. Above twice the critical frequency, the attenuation follows a line with the unity slope in a log-log plot. The function we use is

$$rc(x, x_0, k) = (1 + (x/x_0)^k)^{1/k}, \quad (8.1)$$

where $x$ is the independent variable and $x_0$ and $k$ are parameters. $x_0$ corresponds to the “critical frequency” of a circuit. The response of a real RC circuit would have $k = 2$; here, we control by $k$ the sharpness of the “knee” of the curve. Some normalized cases are plotted in Figure 8.4a. The original optimization cases then correspond to regions in the function (Figure 8.4b). In Region A, sufficient optimization is achieved. Region B contains intermediate cases, and Region C represents weak or missing optimization.

### 8.4.5 Results

To examine the nature of the studied dependency, we calculated the correlation of input and output blow-up separately for each procedure and each of the 143 circuits. The frequencies of correlation values for the simplest and the most powerful procedure are in Figure 8.5. The
8.4. BENCHMARK CIRCUITS IN SOP FORM

\begin{figure}[h]
\centering
\begin{subfigure}{0.49\textwidth}
\includegraphics[width=\textwidth]{figure84a}
\caption{for $k = 0.5, 1, 2$ (highlighted), 5 and 20}
\end{subfigure}\hfill
\begin{subfigure}{0.49\textwidth}
\includegraphics[width=\textwidth]{figure84b}
\caption{with distinguished regions}
\end{subfigure}
\caption{Normalized nonlinear regression function $rc(x, k) = (1 + x^k)^{1/k}$, (Section 8.4.4, page 102)}
\end{figure}

RSS comparison is summarized in Table 8.2. Scatter plots in Figure 8.6 compare RSS values of nonlinear regression with other methods.

Table 8.2: Number of cases, where a particular regression gave the best RSS (Section 8.4.5, page 103)

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Constant</th>
<th>Linear</th>
<th>Linear in log domain</th>
<th>rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>dch if lutpack</td>
<td>0</td>
<td>61</td>
<td>42</td>
<td>39</td>
</tr>
<tr>
<td>20 dch if lutpack</td>
<td>0</td>
<td>77</td>
<td>30</td>
<td>35</td>
</tr>
<tr>
<td>dch if mfs</td>
<td>0</td>
<td>73</td>
<td>45</td>
<td>24</td>
</tr>
<tr>
<td>20 dch if mfs</td>
<td>0</td>
<td>80</td>
<td>53</td>
<td>9</td>
</tr>
</tbody>
</table>

The resulting linear regression parameters are in Figure 8.7 marked by the respective correlation value.

Averaging over 50 input permutations for each synthesis procedure run increased the required computation effort considerably. Histograms in Figure 8.8 illustrate the amount of variance thus eliminated. Isolated cases exist, however, where the variance reached almost 10 for dch if lutpack, and several hundreds for 20 dch if lutpack.

8.4.6 Discussion

High frequencies of high correlation values in Figure 8.5a support Fišer’s conjecture. In Figure 8.7 two classes can be observed. The circuits of the first one have a zero slope and a non-zero intercept, hence they are optimized. The members of the other one have a non-zero slope, often less than unity; they are partly optimized. Bizarre outliers, however, are present.

Available input data did not reveal any behavior that would belong to Region B of the $rc$ function. Furthermore, the nonlinear regression is handicapped by the fact that Region A must lie at the value 1, which is often not true, as presented in Section 8.4.4. Accommodating that feature requires a third parameter to the $rc$ function, which makes the regression process much less stable and robust.
Based on the observations, we may refine Fišer’s conjecture into the following **Nonlinear Conjecture**: We conjecture that the output blow-up depends non-linearly on the input blow-up; furthermore, that the nonlinear dependency combines the cases of successful and unsuccessful optimization, and intermediate cases. Any of those cases may be missing.

The presently used $rc$ regression combines in fact linear regression in the logarithmic domain as asymptotes. As the latter type of regression gives the best fit in less cases than the linear regression, a better nonlinear regression could probably be built with linear dependencies as its asymptotes in Regions A and C.

Collapsing sometimes results in an input blow-up less than one, that is, it sometimes helps to get a smaller description. Synthesis procedures can benefit from this “blow-down” or can produce the same output. $rc$ regression is not always able to fit this behavior. Also, the inability to have a general constant as the Region A asymptote adds to this difficulty.

The variance caused by input permutations was smaller than expected. In Figure 8.8 only variances up to 2 are shown, and yet their frequencies are small.

Comparing `dch if lutpack` with 20 iterations of `dch if lutpack` in Figures 8.5a and 8.8 confirm the findings in Fišer (2012) and Fišer and Schmidt (2012c), that iterating a synthesis procedure trades result quality for result variance.

### 8.5 Synthesis of benchmark circuits after other structure-discarding transformations

Collapsing is not the only way to discard an existing circuit structure. As presented in Section 8.3, global BDDs can play this role as well. BDDs can also be projected directly to structure. Such multiplexer circuits are known to be inefficient (C. Yang, Singhal, and Ciesielski 1999), so this step is useless in practice, but can serve experimental purposes.

#### 8.5.1 Circuits and transformations

The same collection of 490 circuits as in the previous case has been used. A BDD was constructed for each circuit under random variable ordering. A multiplexer structure was derived from the
Figure 8.6: Nonlinear regression for dch if lutpack, RSS comparison; dotted line represents equal RSS (Section 8.4.5, page 103)

Figure 8.7: Linear regression parameters for dch if lutpack (Section 8.4.5, page 103)
(a) for dch if lutpack

(b) for 20 dch if lutpack

Figure 8.8: Variance covered by averaging over input permutations (Section 8.4.5)
BDD in a straightforward manner, using the `dumpBlif()` procedure of the CUDD package. This transformation type will be denoted BDD mux in further text.

Again, blow-ups over 100 were obtained, in several cases even over 1000. Multipliers had again high blow-ups. We managed to process the original $G25$ and $CB(G25)$. Both these circuits exhibited maximum blow-ups as high as the multipliers. Other circuits with known high communication complexity, such as barrel shifters and crossbars, also had high blow-ups. Here, the blow-up is proportional to the size of the BDD, and the facts agree with other results on the BDD size.

The obtained data were tested for representativeness using the same criteria as in the previous experiments; 251 circuits passed the test.

### 8.5.2 Synthesis procedures

As the observed output blow-up variance in the previous experiments was small, we did not cover it by averaging. The following synthesis procedures were used.

- ABC dch if lutpack,
- 20 iterations of ABC dch if lutpack,
- Xilinx XST 9.2 (UG 627)
- Altera Quartus 9.0 (Altera, Inc. 2011)

### 8.5.3 Results

The correlation of the input and output blow-up resulting from the ABC procedures was almost identical to that in the previous experiment. XST and Quartus correlations were similar in nature (Figure 8.9). Similar was also the division of best results into particular regressions, including the $rc$ regression. There was one case, namely the $mux8.128bit$ circuit, where the $rc$ regression yielded substantially lower RSS (0.42 versus 1.44 for linear regression and 4.17 for linear regression in log domain). The data lie in all three regions of the $rc$ model (Figure 8.10a). Other cases of this kind exist for all tested synthesis procedures, however, this one is the most noise-free.
CHAPTER 8. PROBLEMS WITH THE G25 EXAMPLES

Figure 8.10: Examples of synthesis after BDD mux transformation for dch if lutpack

(a) Altera_mux8_128bit, with rc regression
(b) idxlat01n_m12, comparison with synthesis after BDD collapse, regression in the log domain

The fitted models of the synthesis performance on circuits after collapsing and after BDD mux transformation do not correlate. There are circuits having a linear regression slope around one after collapse only, after BDD mux transformations only, both, or none (Figure 8.11). No circuits have a slope much higher than one for both types of transformation. Figure 8.10b illustrates distinct behaviors for each type of transformation.

Figure 8.11: Comparison of linear regression slope for circuits transformed by BDD collapse and BDD mux for dch if lutpack; dotted lines represent equal slope

8.5.4 Discussion
This experiment further supports the Nonlinear Conjecture, and an example where Regions A, B, and C are present. The crucial contribution of this experiment is, that the input blow-up – output blow-up model for a given synthesis procedure depends on the transformation type for
the characterization given – the size.

It remains a future possibility that a certain characteristic will be found such that the model will be the same for all transformations. For now, we are still not able to even characterize practical circuits (Section 4.4), and the hope for such a metric is feeble.

With the present characterization, the results imply that the observed reaction to one transformation cannot be generalized to other transformations. Further implication is, that transformations not occurring in practice cannot be used to evaluate practical performance.

Benchmark circuits must correspond to actual input to logic synthesis (Section 5.4), as there is much less tolerance for inaccuracies and accidental transformations than has been previously thought. When performance of the tools with descriptions of a particular style or expertise level is to be tested, genuine input must be used. In this respect, the examples presented in Fiser and Schmidt (2008) and Fiser and Schmidt (2010) are not much useful.

It may happen that collapsing and similar structure-discarding transformations are a part of a practical design procedure (Section 7.1). In the discussed case, the reason why the description was collapsed is not to give the tool a structural hint, and to force merging the parity tree with the rest of the circuit. Evidently, contemporary tools are not ready to handle such circuit descriptions. The message of the above experiments is, do not use synthesis tools for what they are not intended for. Therefore, the only way to proceed is to use tools that are ready for collapsed descriptions, and those are tools using the classical approach. Of course, these tools have biases as well, and their scalability is certainly a problem.

8.6 Synthesis of benchmark circuits with duplicities

From the previous experiments it follows that the transformation tested so far cannot be used to test the tools’ resistance against incompetent designs. With regard to the above results, such a test would require an authentic material from such users. As a pilot experiment, however, we modeled a frequent design deficiency – not recognizing common functionality in a design.

8.6.1 Circuits, transformations, and synthesis procedures

The same collection of 490 circuits as in the previous case has been used. Transitive fan-ins of branching signals were replicated as described in Fiser and Schmidt (2010), with unlimited depth (i.e. up to primary inputs). 100, 500, 1000, 5000, 10000, and 100000 branches were replicated to obtain descriptions of varying size.

118 circuits had enough data to be evaluated. The set of synthesis procedures was the same as in the previous experiments, except that dch if lutpack was never iterated.

8.6.2 Results

The output of ABC dch if lutpack was not affected by duplication. Therefore, neither iterated runs nor mfs optimization have been measured. For the rest of the procedures, BDD mux transformation and duplication are compared in terms of linear regression slope in Figure 8.12.

8.6.3 Discussion

It can be seen that circuits with duplication are much easier to synthesize than circuits after collapsing or BDD mux transformation. Two reasons are possible:

- This kind of redundancy has been met before, and the algorithms adapted.
- The original structure is not discarded but only enlarged.
Figure 8.12: Comparison of linear regression slope for circuits transformed by BDD mux and duplication; dotted lines represent equal slopes.
Procedures that identify shared logic are present in the classical (Section 5.6.3) and resynthesis (Section 5.8.2) approach. When they were introduced, their usefulness (i.e., area and time gain) has been measured; hence, shared logic must have been present in the examples used for evaluation.

Duplication stresses structural bias of the procedures much less. If there is a feature in the original description, such as some clever decomposition, which the procedure could not re-discover, it is still there after duplication, and does not need to be reinvented.

8.7 Summary

- Structural bias is the principal cause of the G25 results.
- Tools that improve an existing input structure, such as resynthesis-based procedures, are prone to this effect.
- Similar manifestations of structural bias appear in synthesis of other circuits after collapse.
- Other transformations also exhibit a similar influence.
- Characteristics measured by one type of transformation do not in general apply to other transformations.
- Transformations that do not discard the original structure entirely, or which introduce a preconceived redundancy, do not have a comparable effect.
- Synthesis tools depending on input structure must be used within the range of structures that they were intended for. That range should be specified.
- Higher-level synthesis procedures, if they require random logic synthesis, must produce RTL specifications within the given range.
- When greater resistance against structural transformations is required, procedures of the classical approach should be used.
- Synthesis procedures should be evaluated using an authentic material. If the evaluation has less competent designers in mind, it requires a field study and collection of genuine examples.
Chapter 9

Conclusions

We analyzed the problems with synthesis performance reported by Kubalík, Fišer, and Kubátová (2006), and by Cong and Minkovich (2007b). Also, we reviewed the current evaluation practice and circuit examples. We claim the following contributions:

1. We proved that structural bias is the cause of the problems.
2. We modeled how a heuristic behaves under circuit transformations that occurred in the problematic situations.
3. We found an experimental evidence that the model differs between transformation types.
4. We conclude that synthesis algorithms have been (not always consciously) adapted to practical input, and should not be used to process dissimilar inputs, including the problematic examples.
5. We also conclude that the classical (decomposition-based) approach to synthesis, where its scalability allows, is able to overcome the structural bias.

They are described in more detail in the following summary.

9.1 Causes and relevance of the problems

- By principle, heuristic procedures prefer one kind of solution to others; the preference is given by the search algorithm, data representation, data order and other such features.
- Structural bias is the principal cause of the observed problems.
- For such problems to manifest, the synthesis procedure in question must understand the input as a structure (not behavior), and must rely on the input structure.
- Similar manifestations of structural bias appear in synthesis of circuits other than the G25 examples after collapse.
- Other transformations also exhibit a similar influence.
- For a given transformation, the relative change in size is a reasonable characteristic of the transformed input description and of the synthesized output.
- Linear regression in the original or logarithmic domain and the suggested rc function are candidate models of the procedure’s response to the change in the input description.
- Parameters of any model measured by one type of transformation do not in general apply to other transformations.
9.2. **REQUIREMENTS TO AN EFFICIENT SYNTHESIS PROCEDURE**

- Transformations that do not discard the original structure entirely or which introduce a preconceived redundancy, do not have a comparable effect.
- Synthesis procedures of the classical approach have in general small structural bias owing to their input two-level minimization.
- Synthesis procedures of the classical approach can have limited abilities to reconstruct the desired structure; the ability to produce XOR circuits is one.
- Synthesis tools depending on the input structure must be used within the range of structures that they were intended for. When greater resistance against structural transformations is required, procedures of the classical approach shall be used.
- Neither parity predictors nor the examples of Cong and Minkovich detect deficiencies which, if removed, would enhance practical performance of logic synthesis.

9.2 **Requirements to an efficient synthesis procedure**

- Tools based on decomposition must be able to produce a wider class of resulting circuits than it is common now; for example, XOR decomposition is required.
- Synthesis tools depending on input structure must be used within the range of structures that they were intended for. That range shall be specified.
- Higher-level synthesis procedures, if they require random logic synthesis, must produce specifications within the given range.
- When greater resistance against structural transformations is required, procedures of the classical approach shall be used.

9.3 **Requirements to experimental evaluation**

- Practical circuits cannot be characterized in a concise way. Characteristics such as decomposability, short description, group invariants, and design patterns do apply, yet they are insufficient to classify a circuit or to generate artificial examples.
- Languages describing any examples are semantically disparate, and in almost all cases an attempt to describe an example in another language needs design decisions, which loses authenticity.
- Existing benchmark sets are not fully authentic. Unreliable examples include:
  - Synthetic examples form University of Leuven, such as `ex1010`.
  - FSM transition functions with unknown encoding and synthesis, such as `s1488`, `s1494`.
  - Multi-level example processed by inadequate synthesis, the entire multi-level section of LGSynth’93.
  - Newer gate-level benchmarks, if their derivation from the original circuit description is not reproducible.
- To find all sources of variance is still an open problem, even in experimental algorithmics.
- All known sources of variance shall be covered by an evaluation procedure to provide the chosen degree of reproducibility.
- Benchmark circuit sets must document the source of the circuit together with all manipulations with the description, and shall keep those manipulations to a minimum.
- Optimum solutions for problem instances where the circuit is a public benchmark circuit shall be publicly kept.
• The task of logic synthesis does not have a clear, universally accepted definition. In any evaluation, the requirements to logic synthesis shall be stated, e.g. whether the procedure is supposed to synthesize adders.
• To maintain practical significance of experiments, synthesis procedures should be evaluated using authentic material; if the evaluation has less competent designers in mind, it requires a field study and collection of genuine examples.

9.4 Future directions

• The algorithms studied share the following patterns of algorithm design:
  – oblivious and fixed top-level architecture;
  – gradual change in optimization criteria and/or thresholds;
  – topological ordering of nodes within optimization passes.
• There is a possibility that criteria measuring the current circuit description will be developed and used to inform a non-oblivious and intelligent top-level algorithm control.
• The possibility of a synthesis procedure that would be able to handle any description of a circuit (i.e. including transformed descriptions) seems remote.
• Procedures using the classical approach are still worth of research and development, since they have properties that other procedures do not, such as the independence of initial circuit description.


Altera Corp. (Feb. 2008). “Synthesis Design Flows Using the Quartus University Interface Program (QUIP), Version 1.1”. In: Quartus University Interface Program.


BIBLIOGRAPHY


Bollig, B. and I. Wegener (Sept. 1996). “Improving the variable ordering of OBDDs is NP-complete”. In: *Computers, IEEE Transactions on* 45.9, pp. 993–1002. ISSN: 0018-9340. DOI: 10.1109/12.537122.


Corno, F., M. S. Reorda, and G. Squillero (July 2000). “RT-level ITC’99 benchmarks and first ATPG results”. In: Design Test of Computers, IEEE 17.3, pp. 44–53. ISSN: 0740-7475. DOI: 10.1109/54.867894


BIBLIOGRAPHY


Huang, Z. et al. (Dec. 2013). “Fast Boolean matching based on NPN classification”. In: *Field-Programmable Technology (FPT)*, 2013 International Conference on, pp. 310–313. DOI: [10.1109/FPT.2013.6718374](http://dx.doi.org/10.1109/FPT.2013.6718374).


BIBLIOGRAPHY


BIBLIOGRAPHY


Muroga, S. et al. (Oct. 1989). “The transduction method-design of logic networks based on permissible functions”. In: Computers, IEEE Transactions on 38.10, pp. 1404–1424. ISSN: 0018-9340. DOI: [10.1109/12.35838](http://dx.doi.org/10.1109/12.35838)


Index

0-state, 14
1-state, 14
abstraction, 12
abstraction., 8
algebraic, 49
algorithmic level, 8
analysis, 10
analytic, 12
and redundancy removal, 60
And-Inverter-Graph, 17, 59
functionally reduced, 17, 59
structurally hashed, 17, 59
artifact decomposition, 12
ashenhurst decomposition, 22
ashenhurst-curtis decomposition, 50
balancing, 60
base, 17
behavioral, 12
behavioral synthesis, 11
benchmark, see circuit example
benchmark set
LGSynth’93, 64
benchmark sets, 27, 80
Illinois set, 100
ISCAS’85, 100
ISCAS’89, 100
ITC’99, 100
IWLS’2005, 29
LGSynth’91, 27, 100
LGSynth’93, 27, 100
MCNC, 83, 100
bi-decomposition, 48
bias, 35, 55
block, 7
block type, 7
blow-up
input, 100
models, 102, 107, 109
constant regression, 102
linear regression, 102
linear regression in log domain, 102
nonlinear regression, 102
boolean, 49
boolean circuit, 17
Boolean formula, 15
factored form, 16
implicant of, 16, 45
boolean formula, 15, 17
Boolean function, 14
groupable, 48
incompletely specified, 14
multi-output, 15
multivalued, 15
NPN classes of, 18
on-set, off-set, DC-set, 14
representation, 15
canonical, 15
state of, 14
symmetry, 88
boolean function, 14
Boolean network, 16, 45
AIG, 17, 59
BED, 59
BLIF format, 25
cuts in, 17
$K$-feasible, 17
$KL$-feasible, 17
node fan-in, 17
node fan-out, 17
RBC, 59
restricted, 59
boolean network, 16
bound set, 48
canonical, 15

Index:

- cell, 7
- circuit, 7
- circuit example
  - $5xp1$, 85
  - $alu2$, 85, 89, 90, 92
  - $Altera_mux8_{128bit}$, 108
  - $alu1$, 82, 83, 85, 87–92
  - $alu1HFRG$, 86, 88
  - $alu1ITWRC$, 88
  - $alu1QDRG$, 87, 88
  - $alu1RG$, 86–88
  - $alu1TWRG$, 87, 88
  - $alu2$, 85, 87, 89–92
  - $alu3$, 85, 87, 89, 90, 92
  - $alu4$, 28, 29, 63, 64, 85, 89, 90, 92
  - $bw$, 85, 89, 90, 92
  - $CB(G25)$, 95, 97, 99, 107
  - $CD(G25)$, iv, 95, 97, 99
  - collapsing, 100
  - cordic, 28, 29, 74
  - $dalu$, 100
  - $dk17$, 79
  - $dk27$, 79
  - $duke2$, 85
  - $e64$, 85, 89, 90, 92
  - $ex1010$, 31, 113
  - $exp$, 85, 89, 90, 92
  - $f51m$, 88
  - $frg1$, 28, 29
  - $G25$, 95–99, 107, 111, 112
  - $idxlat01n$, 108
  - $ITC_b12$, 100
  - LEKO, 94
  - LEKU, 94
  - $lin$, 85, 89, 90, 92
  - $mainpla$, 56, 85, 89, 90, 92
  - $misc3c$, 85, 87, 89–92
  - $mp2d$, 85, 89, 90, 92
  - $mux8_{128bit}$, 107
  - nasty or nice, 83
  - $opa$, 85, 89, 90, 92
  - parity predictors, 82
  - partial duplication, 109
  - $pope$, 85, 89, 90, 92
  - $prom1$, 85, 89, 90, 92
  - $s1488$, $s1494$, 31, 113
  - $s27$, 28, 29
  - $s298$, 28, 29
  - $s3271$, 54
  - $s420$, 100
  - $s828$, 100
  - $s839$, 100
  - $t1$, 85, 87, 89, 90, 92
  - $t4$, 85, 89, 90, 92
  - too_large, 28, 29
  - transformation, 27, 71, 76, 80, 94, 96, 100, 104, 109
  - $BDD$ mux, 104
  - collapsing, 100
  - partial duplication, 109
  - $XOR$-intensive, 82
  - $z4ml$, 28, 29
  - circuit metric, 96, 100
  - literals, 85
  - $LUT$s, 85
  - collapsing, 51
  - column multiplicity, 49
  - complement, 15
  - constant regression, 102
  - constraints check, 9
  - control part, 8
  - covers, 14
  - custom logic synthesis, 37
  - cut
    - $K$-feasible, 17
    - $KL$-feasible, 17
  - cut, 17, 51
  - data part, 8
  - $dc$-set, 14
  - $dc$-state, 14
  - decomposability problem, 48
  - decomposability wrt. specific function, 48
  - decomposable functions, 22
  - decomposed, 12
  - decomposed function cardinality, 23, 49
  - decomposed function cardinality gain, 49
  - decomposition
    - algebraic, 49
    - Ashenhurst, 50
    - bi-decomposition, 48, 51
    - Boolean, 49
    - by factorization, 50
    - Curtis, 50
    - disjoint, 48
    - functional, 49
INDEX 129

gate-oriented, 48
information-based, 50
of a task, 7, 69
of an artifact, 7
PUB, 50
Roth-Karp, 50
XOR-capable, 89, 91
decomposition, 47
design, 7
abstraction, 8
analytic and synthetic steps, 9
decomposition, 7
domain analysis, 9
disjoint, 16
disjoint decomposition, 22
disjoint sum of products (DSOP), 16
distance-\(k\), 17
distance-\(m\) transitive fan-in, 17
distance-\(m\) transitive fan-out, 17
domain
transitions, 18, 33, 34
dominator, 51

EDA command
-xhardcore, 89
-xhc, 89
ABC balance, 96
ABC collapse, 96
ABC dsd, 98
attribute foreign, 38
b, 62
balance, 62, 94
BDD collapse, 108
BDD mux, 107–110
BDS -useglb, 89
BDS -useglb -xhc, 89
choice, 61, 63
choice fpga lutpack, 83
choice; fpga; ps, 63
collapse, 86
collapse, 54, 89, 100
dch, 61, 63, 89, 100
dch if lutpack, 28, 74, 101, 103–109
dch if mfs, 101, 103, 104
dsd, 96, 97
dsd + fpga, 99
dumpBlif(), 107
EXPAND, 52
EXPAND_GASP, 53

fpga, 63, 96, 99
gcx, 51
gkx, 51
if, 61, 62, 89, 100
IRRENDUNDANT, 53
LAST_GASP, 53
lutpack, 89, 100
mfs, 60, 63, 100, 109
permute, iv
plus, 38
pragma map_to_operator, 38
print_symm, 88
REDUCE, 36, 52
REDUCE_GASP, 53
resyn, 61
resyn2, 96
resyn2 + fpga, 99
rewrite, 61
script-algebraic, 54
script_rugged, 83
SIS tech_decomp, 96
st; dch; if {C 12; mfs {W 4 {M 5000, 63
strash, balance, fpga, 96
tech_decomp, 92, 94, 96, 97
tech_decomp -a 4, 83, 84

EDA tool
ABC, 33, 61, 62, 83, 96
Altera Quartus II, 107
BDS, 34, 51, 89, 96
BDS-MAJ, 51
BDS-PGA, 51
BiDecomp, 96
block generator, 38
BOOM, 47
Design Compiler, 45
ESPRESSO, 46, 52
LSS, 58
MIS, 45
MVSIS, 86
QUIP, 34
SIS, 45, 49, 83, 86, 91, 96
SOCRATES, 58
Xilinx Coregen, 38
Xilinx XST, 107
elimination, 51
encoding, 15
encoding problem, 48
entity, 7
INDEX

experiment
design, 67
interpretation, 67
robustness, 84, 85
with parity predictors, 82
experimental algorithmics, 66, 70
experimental evaluation, 67
error elimination, 68, 73
experiment design, 71
qualitative, 70
quantitative, 69
source of variance in, 73
experiments
with circuits in SOP, 98
with discarded structure, 104
with duplicities, 109
with G25, 96
extraction, 51

factored form, 16
factored form, 16, 50
factoring, 47
factorization, 50
factorization, 50
fan-in, 17
fan-out, 17
Fíšer’s conjecture, 101
flattening, 51
free set, 48
function representation, 34
functional decomposition, 49
functional separability, 22
functionally reduced aig (fraig), 17
gate decomposition, 48
gate level, 8
generalized dominator, 51
global methods, 8
glue logic synthesis, 37
group invariance, 22
groupable, 48

hardware description languages
EDIF, 27
elaboration, 37

hardware description languages, 24
ABEL, 37, 45
BENCH, 25
BLIF, 25
conversions, 25

EDIF, 25
elaboration, 29
synthesis from, 37
Verilog, 24
VHDL, 25
hierarchy, 7, 12
higher-level synthesis, 11, 13
implementation, 7
implicant, 16
implicant, 16
incompletely specified, 14
incompletely specified multivalued, 15
input blow-up, 100
input cone, 17
input nodes, 16
instance, 7
integration, 8
intersection, 15

leaf set, 17
linear regression, 102
linear regression in log domain, 102
literal, 16
literal, 16
logic synthesis, 33
adders, 40
bias in, 35, 55
classical approach, 44
decomposition step, 45, 48, 89
domains in, 34
function representation in, 34
multi-level minimization, 45
multi-level optimization, 47
random (custom, glue) logic, 37
resynthesis approach, 59
rule-based, 57
scalability, 56
two-level minimization, 44, 45
logic synthesis, 11, 13, 37

memory locations, 8
metric
circuit complexity of, 85
on a set of states, 19
on a state space, 20
module, 7
multi-output boolean function, 15
multi-output cut, 17
multivalued, 15
INDEX

multivalued boolean function, 15
multivalued function, 50

near symmetry, 88
node function, 16
nonlinear conjecture, 104
nonlinear regression, 102
NPN class, 18
NPN equivalence, 18

off-set, 14
on-set, 14
operational units, 8
optimization algorithm
  bias in, 35
  global heuristic, 55
  randomized, 35
  state space, 20
output blow-up, 100
output cone, 17
output nodes, 16

parallel, 12
parallel task decomposition, 8
parity predictor, 82
partial symmetry, 22
partitioning problem, 48
phases, 8
physical, 12
physical design, 13
PLA form, 16, 25, 42
practical, 24
practical circuits, 21, 66, 69, 94, 109
  authenticity, 70
  decomposability, 22
  examples
    authenticity, 25
    benchmarks, 84
    nastiness, 84
    parity predictors, 84
    invariants, 22
    patterns, 23
    representations, 24, 34
predecessor function, 48
product term, 16
random combinational logic synthesis, 37
random logic synthesis, 37
randomization, replication, and
  organization to reduce error, 68
rc-curve, 102
register transfer level, 8
representation, 15
represents, 15
resubstitution, 51, 60
rewriting, 60
root, 17
root set, 17

Satisfiability Problem, 20
scalability, 56
selection, 15
serial, 12
serial task decomposition, 8
simple dominator, 51
specification, 7
state
  of a Boolean function, 14
  of a local search algorithm, 18
  metric set, 19
state, 14
state space
  of a local search algorithm, 20
    symmetric move structure, 20
strongly groupable, 48
structural, 12
structurally hashed AIG, 17
substitution, 51
successor function, 48
suitability check, 9
sum of products
  disjoint, 16
  sum of products (SOP), 16
sum term, 16
symmetric move structure, 20
symmetry
  near, 88
  symmetry, 22
synthesis, 10
synthetic, 12
task decomposition, 12
task decomposition and hierarchy, 7
technology mapping, 33, 43
technology mapping, 10, 13
transistor level, 8
transitive fan-in, 17
transitive fan-out, 17
trivial, 50
union, 15
vector boolean function, 15
verification, 9
views, 25

weakly groupable, 48

Y diagram, 9
  in synthesis and mapping, 34
  transitions in, 18, 33, 34