SAT-Based Exact Synthesis for Multi-Level Logic Networks

Thèse N° 9404

Présentée le 31 mai 2019

à la Faculté informatique et communications
Laboratoire des systèmes intégrés (IC/STI)
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Winston Jason HAASWIJK

Acceptée sur proposition du jury

Prof. P. Ienne, président du jury
Prof. G. De Micheli, Dr M. Soeken, directeurs de thèse
Dr A. Mishchenko, rapporteur
Prof. P. Beerel, rapporteur
Prof. D. Atienza, rapporteur

2019

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Beware of bugs in the above code;
I have only proved it correct, not tried it.
— Donald Knuth

To my family.

Acknowledgements

Writing a PhD thesis is not an easy task. It is clear to me that this work would not have been possible without the kind help of many people. Unfortunately, there is no way for me to do justice to all of them here. Nevertheless, I will do my best.

First, I would like to thank all of my friends and colleagues at the LSI. The relaxed and friendly atmosphere at the office made a big difference in my PhD experience, especially knowing that not everyone is so lucky.

Second, I want thank my original co-advisor Pierre-Emmanuel Gaillardon. It was PE who showed me how the academic sausage is made when I first came to EPFL.

Next, I would like to mention some of our friends and partners from Synopsys Inc. for being such gracious hosts and supporters of my work. I especially thank Luca Amarù, Patrick Vuillod, Jiong Luo, and Janet Olson.

I have to mention and thank Alan Mishchenko for our many valuable discussions. A true logic synthesis guru, our conversations were always inspiring. Afterwards I would be excited to start playing around with new ideas.

Academic research is a skill, and I have not received more practical tips and tricks about it from anyone else than Mathias Soeken. I am indebted to him for countless ideas, discussions, and collaborations. Without his help my time in Switzerland would have been much more difficult.

Speaking of research, there is one last academic I have not mentioned yet. I sincerely thank Giovanni De Micheli for allowing me to do my PhD research at the LSI. Nanni was always patient and gave me the time, space, and freedom to pursue the projects that I was interested in. I could not have asked for more.

My time in Switzerland would not have been the same without a great social support system. I would like to thank all of my friends in Switzerland, from EPFL and beyond. Thanks for making this time such a great adventure.

Finally, I want to express my deep thanks and gratitude to my family. Knowing that I could always come home to recharge was a great comfort to me. I am sure that it helped me to reach the end goal. Thank you for your support and for your patience.

Thank you all.

Jerusalem, April 21, 2019 Winston J. Haaswijk

Preface

How many two-input gates are needed in a logic network to compute an arbitrary six-input Boolean function? How many majority-of-three operations suffice to compute the majority of nine Boolean variables? Precise answers to these seemingly tractable questions surprisingly do not exist. Exact synthesis describes a practical approach to tackle these difficult problems. Answers to the above and many more similar other questions have significant impact to state-of-the-art logic synthesis applications. However, until recently exact synthesis algorithms have mainly been used in order to strengthen theoretical upper bounds on circuit complexity results. Their intensive run-time requirements have rendered them infeasible in a robust scalable logic optimization flow.

In this thesis, Winston has thoroughly investigated exact synthesis and unfolded the existing proof-of-concept SAT-based algorithms into a robust methodology that enables practical application. To this end, he proposes solutions from various different angles. He motivates the careful study of encodings and symmetry breaking techniques in an elaborated experimental evaluation that compares various encodings—including a new one proposed in this thesis—and various symmetry breaking constraints.

The strongest contribution by Winston begins with the observation that exact synthesis essentially solves the two tasks of finding a logic network topology and assigning gate operators at the same time. The thesis shows that separating these tasks can significantly simplify the complexity of the problem and therefore lead to tremendous run-time improvements. The two tasks can be separated by preassigning topology information of the logic network. Such separations know two extremes: (i) not assigning any topology information which results in a single monolithic SAT problem which is difficult to solve; or (ii) assigning full topology information which results in many simple and easy to solve SAT problems, one for each possible topology—a number that quickly explodes already for a small number of gates.

Winston proposes two alternatives for non-trivial partial topologies that suggest a good compromise between the number of subproblems and their complexity. These are fences and partial dags. Both come with theory, practical algorithms, and experimental evaluations. Further, the partition into subproblems naturally enable parallel execution, which outperforms parallel SAT solving techniques that are not aware of the topology separation in the problem.

All techniques that are described in this thesis are implemented in the open source C++ library percy. The quality of the library is quite remarkable. It will help researchers to extend the

state-of-the-art in SAT based exact synthesis and help practitioners to easily evaluate the impact of exact synthesis in existing logic optimization flows. This thesis marks the end of the fruitful PhD studies of Winston and the beginning of the field of practical exact synthesis techniques. They will shape the field of logic synthesis as did the emergence of algebraic and Boolean techniques in the past.

Montreux, April 21, 2019 Mathias Soeken

Abstract

Today, the design of electronic systems is largely automated. The practice of using software automation technologies for the design of electronic hardware is commonly referred to as Electronic Design Automation (EDA). EDA comprises a large set of tools, from languages that specify high-level hardware designs, to software that determines the layout of nanoscale devices on an integrated circuit. Within this collection, an important role is played by so-called *logic synthesis* algorithms. A substantial field of research onto itself, logic synthesis can be roughly thought of as the problem of finding good representations for Boolean functions. Such functions are the backbone of digital circuits, which can be thought of, to a first approximation, as devices that compute with Boolean values. In other words, circuits can be viewed simply as large Boolean functions. Logic synthesis, then, is assigned the important task of finding good structural representations for such circuits. Choosing the right logic synthesis algorithm can have a significant impact on the efficiency of an electronic design.

Depending on one's reckoning, the development of logic synthesis tools can be traced back to Claude Shannon's famous 1937 master's thesis on switching circuits, or perhaps even further back to George Boole's seminal work from the 1800s on what is now known as Boolean algebra. Development of multi-level logic synthesis started some decades ago, in the late 1970s. Due to the complexity of multi-level logic synthesis, current algorithms are mostly based on heuristic methods.

In this thesis, we consider a special type of logic synthesis algorithm known as exact synthesis. Specifically, we analyze and develop multi-level exact synthesis algorithms based on a SAT formulation. Such algorithms attempt to solve a very difficult problem in which, given any Boolean function, they find the optimum (i.e. best possible) circuit that represents it. Our contributions can be roughly split into two parts: (i) core exact synthesis algorithms, and (ii) applications of exact synthesis. In the first part, we start by examining, in detail, different ways to encode the exact synthesis problem into CNF formulas. These formulas are given as input to SAT solvers, which solve them to find solutions. Finally, the solutions to these formulas can be decoded to obtain optimum Boolean circuits. We will compare and contrast the different encodings and show, quantitatively and experimentally, that a proper encoding can greatly influence the efficiency of exact synthesis algorithms. Next, we will show how such exact synthesis algorithms can be improved by adding domain-specific information. This information takes the form of families of DAG topologies which contain some additional

structure to guide the SAT solver in its search. Furthermore, using these DAG topology families, we show how the exact synthesis problem can be transformed into an embarrassingly parallel one. This essentially means that, as long as we have processors available, we can throw more and more parallel computing power at the problem to solve it more quickly. After analyzing and improving the core exact synthesis algorithm, we arrive at the second part of the thesis. In this part, we show how exact synthesis can be applied to different problems that are of both theoretical and practical interest. On the theoretical side, we show how exact synthesis can be used to classify Boolean functions in terms of their intrinsic difficulty. On the practical side, we introduce a new data structure and logic representation called XOR-Majority Graphs (XMGs). We use XMGs, in concert with exact synthesis, to develop a novel logic rewriting methodology which achieves significant improvements over the state-of-the-art. We then generalize this methodology into one that can be used to restructure arbitrary Boolean networks, again enabling new improvements in logic synthesis.
Overall, the contributions of this thesis show how exact synthesis can be improved and applied in the modern age of parallel hardware and software. We expect that it will be an essential part of any EDA toolbox in the years to come, during which design goals are likely to become even more ambitious than they have been in the past.

Keywords: electronic design automation, EDA, logic synthesis, exact synthesis, Boolean satisfiability, SAT, SAT solvers, formal methods.

Zusammenfassung

Der Entwurf elektronischer Systeme ist heutzutage weitestgehend automatisiert. *Electronic Design Automation* (engl., etwa elektronische Entwurfsautomatisierung) ist das Gebiet, welches automatische Softwarelösungen zum Entwurf elektronischer Hardware verwendet. EDA beinhaltet eine große Anzahl an Werkzeugen, beginnend von Sprachen zur abstrakten Beschreibung von Hardwaremodellen, bis hin zu Software, die das konkrete Layout von Bauteilen im Nanobereich auf dem integrierten Schaltkreis berechnet. Als substantielles Teilgebiet beschreibt Logiksynthese das Problem gute Darstellungen für Boolesche Funktionen zu finden. Boolesche Funktionen sind essentiell für digitale Schaltkreise, welche als elektronische Systeme verstanden werden können, die mit Booleschen Wahrheitswerten rechnen. In anderen Worten können Schaltkreise auch als große Boolesche Funktionen verstanden werden. Die wichtige Aufgabe von Logiksynthese ist gute strukturelle Darstellungen für solche Schaltkreise zu finden. Die Wahl des Logiksynthesealgorithmus hat einen signifikanten Einfluss auf die Effizienz des zu entwerfenden elektronischen Systems.

Die Entwicklung von Logiksynthesewerkzeugen geht zurück auf Claude Shannon's berühmte Masterarbeit über *Switching Circuits*, vielleicht sogar auf George Boole's einführende Werke aus dem 19. Jahrhundert, die mittlerweile als Boolesche Algebra bekannt sind. Die Entwicklung effizienter Softwaretools begann in den späten 70er Jahren des letzten Jahrhunderts. Wegen der hohen Komplexität von Logiksynthesealgorithmen, basieren fast alle existierenden und derzeit verwendeten Methoden auf Heuristiken.

Diese Dissertation betrachtet *exakte* Synthese, eine spezielle Art von Logiksynthese. Wir betrachten insbesondere SAT-basierte exakte Synthesealgorithmen. Diese Algorithmen versuchen ein besonders schweres Problem zu lösen: finde die beste Darstellungen zu einer beliebigen Booleschen Funktion. Unser Beitrag umfasst grob zwei Teile: (i) Kernalgorithmen für exakte Synthese und (ii) Anwendungen exakter Synthese. Im ersten Teil untersuchen wir im Detail unterschiedliche Arten exakte Synthese als KNF Formeln zu kodieren. Diese KNF Formeln sind Eingaben für SAT Beweiser, welche erfüllende Lösungen zu den Formeln finden. Schaltkreise können aus den Lösungen dekodiert werden. Wir vergleichen verschiedene Kodierungen und stellen sie gegenüber, und zeigen quantitativ sowie experimentell, dass die Wahl der Kodierung einen starken Einfluss auf die Effizienz exakter Synthesealgorithmen haben kann. Anschließend zeigen wir wie exakte Synthesealgorithmen durch das Hinzufügen von domänenspezifischen Informationen verbessert werden können. Diese Informationen

beinhalten Mengen von Graphstrukturen, welche den Suchraum für die SAT Beweiser signifikant reduzieren können. Wir zeigen außerdem wie diese Graphstrukturen zur effizienten Parallelisierung des SAT Beweisers verwendet werden können. Im zweiten Teil der Dissertation zeigen wir wie exakte Synthesealgorithmen für verschiedene Probleme von praktischem und theoretischem Interesse verwendet werden können. Wir zeigen eine theoretische Anwendungen zur Klassifizierung Boolescher Funktionen basierend auf ihrer Komplexität. Als praktische Anwendung zeigen wir einen Optimierungsalgorithmus für XOR-Majority Graphen (XMGs), eine neue Logikrepräsentation. Die Ergebnisse dieses Optimierungsalgorithmus zeigen signifikante Verbesserungen gegenüber existierender Methoden. Zudem haben wir den Optimierungsansatz für generelle Boolesche Logiknetzwerke verallgemeinert.
Zusammenfassend zeigen die Beiträge dieser Dissertation wie exakte Synthese unter der Verwendungen moderner paralleler Hardware und Software eingesetzt sowie verbessert werden. Wir gehen davon aus, dass die vorgestellten Techniken ein essentieller Bestandteil zukünftiger EDA Softwarelösungen sein werden, um die immer zu ambitionierteren Hardwaresysteme effizient entwerfen zu können.

Schlagworte: Elektronische Entwurfsautomatisierung, Logiksynthese, exakte Synthese, Boolesches Erfüllbarkeitsproblem, SAT Beweiser, Formale Methoden

Samenvatting

Het ontwerp van elektronische systemen is tegenwoordig grotendeels geautomatiseerd. De term *Electronic Design Automation* (EDA, Nederlands: elektronische ontwerpautomatisering) verwijst naar de tak van de informatica die zich bezighoudt met het ontwikkelen van softwareoplossingen voor het ontwerpen van elektronische hardware. EDA omvat een groot aantal hulpmiddelen, van talen waarmee we op een hoog niveau het gedrag van hardwareontwerpen kunnen omschrijven, tot software die op nanoschaal de concrete opmaak bepaald voor de bouwstenen van elektronische circuits. Een belangrijke onderdeel van EDA zijn de zogenaamde *logicasynthese*-algoritmes. Logicasynthese is een substantieel onderzoeksgebied waarin men probeert goede representaties van Booleaanse functies te vinden. Zulke functies zijn essentieel voor digitale apparaten, aangezien we deze kunnen beschouwen als apparaten die rekenen met Booleaanse waarden. Anders gezegd kunnen we digitale circuits simpelweg beschouwen als grote Booleaanse functies. Het doel van logicasynthese is het vinden van goede structurele representaties voor zulke circuits. De keuze voor het juiste logicasynthesealgoritme kan van grote invloed zijn op de efficiëntie van een elektronisch systeem.

De ontwikkeling van logicasynthesealgoritmes is terug te leiden naar Claude Shannon's beroemde masterscriptie uit 1937, of zelfs nog eerder naar het baanbrekende werk van George Boole uit de 19^{e} eeuw waaruit het idee van de Booleaanse Algebra is ontstaan. De ontwikkeling van meer geavanceerde (en geautomatiseerde) algoritmes begon laat in de jaren zeventig van de 20^{e} eeuw. Vanwege de enorme complexiteit van het fundamentele logicasyntheseprobleem zijn huidige algoritmes grotendeels gebaseerd op heuristieke methodes.

In dit proefschrift onderzoeken we een speciaal soort logicasyntheseprobleem dat bekend staat als *exacte* logicasynthese. We analyseren en ontwikkelen verschillende multi-level exacte logicasynthesealgoritmes die gebaseerd zijn op een SAT formulering. Deze algoritmes proberen een bijzonder moeilijk probleem op te lossen. Namelijk, gegeven een Booleaanse functie vinden zij het optimale (best mogelijke) circuit voor die functie. Onze bijdragen in dit proefschrift kunnen grofweg worden opgedeeld in twee onderdelen: (i) kernalgoritmes voor exacte logicasynthese en (ii) toepassingen van exact synthese. In het eerste gedeelte onderzoeken we verschillende manieren om het exacte syntheseprobleem te coderen als CNF formules. Deze formules worden aan SAT-oplossers ingevoerd die we vervolgens gebruiken om oplossingen te vinden. Tenslotte kunnen de oplossingen van deze formules gedecodeerd

worden om optimale circuits te vinden. We zullen verschillende coderingen met elkaar contrasteren en vergelijken. We zullen ook laten zien dat de keuze voor de juiste codering van grote invloed kan zijn op de efficiëntie van exacte synthesealgoritmes. Vervolgens laten we zien hoe deze algoritmes kunnen worden verbeterd door de toevoeging van domeinspecifieke informatie. Deze informatie neemt de vorm aan van families van DAG-topologieën welke extra structuur aanbrengen die de SAT-oplosser helpt bij het zoeken naar oplossingen. Bovendien kunnen deze DAG-topologieën gebruikt worden om parallelle exacte synthesealgoritmes te ontwikkelen. In essentie betekent dit dat, zolang we meer processoren tot onze beschikking hebben, we altijd meer computerkracht kunnen toevoegen om het probleem op te lossen. Na het analyseren en verbeteren van de kernalgoritmes komen we bij het tweede gedeelte van dit proefschrift. Hierin laten we zien hoe exacte synthese kan worden toegepast op verschillende problemen die zowel theoretisch en praktisch van belang zijn. Aan de theoretische kant laten we zien hoe exacte synthese gebruikt kan worden om Booleaanse functies te classificeren aan de hand van hun intrinsieke complexiteit. Aan de praktische kant introduceren we een nieuwe datastructuur en representatie: de zogenaamde XOR-Majority Graphs (XMGs). We gebruiken de combinatie van exact synthese en XMGs om een nieuw *logica-herschrijvend* algoritme te ontwikkelen dat aanzienlijk betere resultaten behaald ten opzichte van bestaande algoritmes. Daarna laten we zien hoe dit algoritme gegeneraliseerd kan worden. De generieke versie van ons algoritme kan worden gebruikt om verschillende soorten circuits te verbeteren, hetgeen een belangrijke stap is in de vooruitgang van dit type logicasynthesealgoritmes.
De bijdragen van dit proefschrift laten zien hoe exacte synthesealgoritmes verbeterd en toegepast kunnen worden in de moderne tijd van parallelle hardware en software. Onze verwachting is dat zij in de komende jaren een essentiële rol zullen spelen in veel EDA-oplossingen, vooral aangezien de ontwerpdoelen van hardwarefabrikanten alsmaar ambitieuzer worden.

Trefwoorden: elektronische ontwerpautomatisering, EDA, logicasynthese, exact synthesis, vervulbaarheidsprobleem, SAT, formele methoden.

Contents

Acknowledgements **v**

Preface **vii**

Abstract (English) **ix**

Abstract (Deutsch) **xi**

Abstract (Nederlands) **xiii**

List of figures **xvii**

List of tables **xx**

1 Introduction **1**
1.1 Boolean Functions . . . 3
1.2 Logic Synthesis . . . 4
1.2.1 Exact Synthesis . . . 10
1.3 Motivation . . . 11
1.4 Thesis Contributions . . . 13
1.4.1 Encodings & Quantitative Comparisons . . . 14
1.4.2 DAG Topology Families . . . 14
1.4.3 Parallel Synthesis . . . 14
1.4.4 Applications . . . 15
1.5 Thesis Outline . . . 15

I Core Algorithms **19**

2 Synthesis & Encoding **21**
2.1 Background . . . 22
2.1.1 Boolean Chains . . . 22
2.1.2 SAT-based Exact Synthesis . . . 24

2.1.3 A Note on Optimality . . . 26
2.2 CNF Encodings . . . 26
2.2.1 Single Selection Variable (SSV) Encoding . . . 27
2.2.2 Multiple Selection Variables (MSV) Encoding . . . 30
2.2.3 Distinct Input Truth Tables (DITT) Encoding . . . 31
2.3 Symmetry Breaking . . . 34
2.4 Quantitative Comparisons of CNF Encodings . . . 37
2.5 CEGAR . . . 42
2.6 Synthesis With Don't Cares . . . 45
2.7 Computational Complexity . . . 46
2.8 Summary . . . 50

3 DAG Topology Families **51**
3.1 Introduction . . . 51
3.2 Fences . . . 52
3.3 Partial DAGs . . . 53
3.4 Counting Dags, Fences, and Partial DAGs . . . 55
3.5 Generating Fences . . . 57
3.5.1 Integer Partitioning Method . . . 57
3.5.2 Recursive Backtracking Method . . . 58
3.6 Exact Synthesis Using Fences . . . 60
3.7 Fence vs. Conventional Encodings . . . 63
3.8 Synthesis With Partial DAGs . . . 64
3.9 Topology-Based Parallel Exact Synthesis . . . 65
3.10 Topology-Based vs. Generic Parallelism . . . 66
3.11 Majority-7 Decomposition . . . 68
3.12 Summary . . . 69

II Applications **71**

4 Function Classification **73**
4.1 Introduction . . . 73
4.2 NPN Canonization . . . 76
4.3 Classification Method . . . 78
4.3.1 Finding All NPN Classes . . . 78
4.3.2 Finding Minimum-Size Chains With Exact Synthesis . . . 79
4.3.3 Synthesis Upper Bounds . . . 82
4.4 Experimental Results . . . 82
4.5 Summary . . . 84

5 Optimizing XOR-Majority Graphs **85**
5.1 Introduction . . . 85
5.2 Preliminaries . . . 86
5.2.1 Cut Enumeration . . . 86
5.2.2 Logic Rewriting . . . 89
5.2.3 LUT Mapping . . . 94
5.3 Contributions . . . 95
5.4 XOR-Majority Graphs . . . 96
5.5 Optimization Method Overview . . . 98
5.5.1 Comparison to Previous Work . . . 99
5.6 Method Implementation . . . 100
5.6.1 Exact Synthesis . . . 100
5.6.2 XMG Size Optimization . . . 103
5.7 Experimental Evaluation . . . 103
5.7.1 XMG Size Optimization . . . 104
5.7.2 LUT Mapping . . . 105
5.7.3 Comparison To Best Known Results . . . 106
5.8 Summary . . . 107
5.8.1 Future Work . . . 108

6 Optimizing Boolean Networks **109**
6.1 Introduction . . . 109
6.2 Preliminaries . . . 110
6.3 Cut Rewriting . . . 111
6.3.1 Efficiency Tricks & Don't Cares . . . 113
6.4 Experiments . . . 114
6.5 Summary . . . 116

7 Conclusions **117**
7.1 Thesis Contributions . . . 118
7.2 Future Work and Open Problems . . . 120
7.3 Final Remarks . . . 121

A *percy*: an exact synthesis library **123**
A.1 Code Examples . . . 124
A.2 A Note on Correctness . . . 129

Bibliography **131**

Curriculum Vitae **145**

List of Figures

1.1 A conceptual representation of some important complexity classes. 3
1.2 Example of a simplified EDA flow. 5
1.3 Schematic illustration of a simple NMOS PLA. Courtesy of Professor Jan M. Rabaey. 7
1.4 Illustration of a bound heterogeneous Boolean logic network for a full adder. . 9
1.5 A logic rewriting flow. Optimization of the subnetworks may be achieved in various ways, such as database retrieval, heuristic decomposition, or exact synthesis. 12

2.1 Illustration of an optimum Boolean chain for a full adder. 23
2.2 Illustration of a basic size-optimum SAT-based exact synthesis algorithm. . . . 25
2.3 Examples of circuit topologies that are avoided by applying symmetry break **(A)**. 35
2.4 Illustration of the kinds of circuit structures avoided by symmetry break **(R)**. . . 35
2.5 Illustration of symmetries avoided by the (co)-lexicographical symmetry break. Using **(C)**, the topology in (a) would not be valid, whereas the one in (b) would be. 36
2.6 Illustration of symmetries avoided by symmetry break **(S)**. In the topology on the left, we have switched variables x_3 and x_4. Although both topologies are nearly identical minimum-size implementations of f, (a) is invalid under **(S)**, since x_3 must be used before x_4. 37
2.7 An illustration of the impacts of CEGAR on synthesis runtime. 45

3.1 Illustrations of the first five fence families. 54
3.2 On the left an example of partial DAG specified by the sequence below. Unspecified fanins are signified by empty circles. On the right a fully specified chain found by the SAT solver for the function $f = \langle x_1 x_2 x_3 \rangle$. 55
3.3 The fence F in (a) corresponds to a set of possible DAG topologies and can thus be used to constrain the SAT solver's search. For instance, Figure (b) and Figure (c) satisfy the constraints from F. Figure (d) does not. Each node on level λ must have at least one fanin from level $\lambda - 1$; this follows by definition of levels. 62
3.4 Shows, for a set of 500 hard benchmarks, the number of successfully synthesized chains within the 1 minute timeout. 65
3.5 Shows how topology information may be used to create an embarrassingly parallel exact synthesis pipeline. 66

3.6 A comparison between our domain-specific parallelism and a generic parallel SAT backend. 67
3.7 Illustration of the super-linear speedup achievable by topology-based parallel synthesis. 68
3.8 Comparison of majority-7 decomposition between the best SSV encoding and a fence-based encoding with an increasing number of threads. 69

4.1 An example of two different functions that are P-equivalent. The circuit in (a) can be made equivalent to the one in (b) by permuting the inputs. 76
4.2 An example of two different functions that are NPN-equivalent. The circuit in (a) can be made equivalent to the one in (b) by negating its output and permuting the inputs. 76
4.3 We can implement any 4-input Boolean function by using at most three 3-input operators. 82

5.1 Estimating the gain of a replacement cut using reference counting. 91
5.2 Any k-feasible cut can be implemented by a single k-LUT. In this example $k = 3$. 94
5.3 Size-optimum full adders, given in AIG, MIG, and XMG representations, respectively. Dashed lines indicate complemented edges. We see that $\sigma(a) \leq \sigma(b) \leq \sigma(c)$. 97
5.4 An overview of the optimization flow. 98

7.1 Even without an exponential speedup, we can make SAT-based exact synthesis more practical through techniques such as symmetry breaks and topology-based parallelism. 118

List of Tables

1.1 Notation for some commonly used Boolean operators. 5

2.1 Impact of symmetry breaking on the space of 4-input functions for 2-input operator chains. Sorted by average synthesis time. All times reported in μs. . . 40
2.2 Impact of encoding and symmetry breaking for 5-input functions with 3-input operator chains. 41
2.3 Impact of encoding and symmetry breaking for 6-input functions with 4-input operator chains. 42

3.1 Comparing the numbers of DAGs, partial DAGs, and fences for increasing numbers of vertices. 57
3.2 Comparing fence- and partial DAG-based synthesis to conventional state-of-the-art encodings. All runtimes in ms. 64

4.1 Combinational complexity of all 4-input functions using 2-input operators [69] 75
4.2 Combinational complexity of all 5-input functions using 2-input operators [69] 75
4.3 Comparing the number of n-input functions and NPN classes. Numbers of NPN classes taken from [127]. We write the numbers for $n = 8$ in scientific notation, as they would not fit on the page otherwise. 78
4.4 Combinational complexity of all 4-input functions using 3-input operators . . 83
4.5 Combinational complexity of all 5-input functions using 3-input operators . . 84

5.1 Comparing XMG and AIG size optimization. 105
5.2 Comparing 6-LUT Mapping for XMGs and AIGs 106
5.3 Comparing Best XMGs To Best Known 6-LUT Mapping Results. 107

6.1 Cut rewriting experimental results for 3-LUT resynthesis 115
6.2 Cut rewriting experimental results for 4-LUT resynthesis 115

1 Introduction

Electronic systems are ubiquitous, and their presence in the world is only increasing. Moreover, there is a trend towards increasing design complexity across multiple levels of abstraction. For example, on the device level, novel technologies such as quantum-dot cellular automata and spin-wave devices pose different problems than those encountered in conventional CMOS [80, 120]. Designs using such technologies must make efficient use of novel logic primitives, while accounting for complex constraints such as fanout or depth restrictions [141]. On the architectural level, there has been a trend towards *system on a chip* (SoC) design. SoCs integrate various components such as CPUs, memories, video decoders, and sensors on a single substrate [109]. Communication between SoC components is challenging and has led to the so-called *network on chip* (NoC) design paradigm [16]. With the advent of the *Internet of Things* (IoT), many devices must be able to sense, signal, or otherwise interact with their environments, often under tight energy constraints [145]. The design of IoT systems is particularly challenging as it involves, besides issues such as device and network architecture, also far-reaching privacy and security concerns [73]. Finally, to remain economically viable, reducing time-to-market is an important design target for many systems. Introducing new products faster than the competition has many advantages: it affects both cost and potential market value, allows companies to set technical standards, and enables them to respond more quickly to customer feedback [149].

Given the complexity of current systems, it is no longer feasible to create full error-free designs in a cost-effective time frame, even by large teams of human designers. Indeed, this has not been feasible for several decades [88, p. 5]. The goal of *electronic design automation* (EDA) is to aid human designers by providing techniques to automate large parts of the design process. Over the years, sophisticated methods for synthesis, test, and verification have been developed. These are applicable to various design stages, such as architecture level synthesis, logic synthesis, and physical design. In this thesis, we analyze and develop logic synthesis techniques based on *Boolean satisfiability* (SAT).

The *Handbook of Satisfiability* identifies two key roles of SAT: (i) reasoning about propositional logic formulas, and (ii) solving combinatorial problems [19, p. v]. Propositional logic is the study of propositions that may be *true* or *false*, as well as the logical connectives that are used to compose them. SAT refers to the problem of determining whether or not a proposition can ever by true (i.e. satisfied). We can use SAT to elegantly describe problems in declarative way. Furthermore, it serves the role of an efficient and generic computational substrate.

We are often faced with the task of deciding if some combinatorial object has a certain property. Such tasks are known as *decision problems*. Now, one might object that the notions of "combinatorial object" and "property" are ill-defined here. However, this is not without reason: their rigorous definition requires quite some work. This becomes perhaps easier to understand when one imagines the wide range of possibilities. For example, we may want to know if a graph can be colored with a certain number of colors, or if we can pack a knapsack while not exceeding a certain weight. To avoid being bogged down in technical details, we allow ourselves to be less than rigorous here. We do not require the reader to be deeply familiar with the theory of computation, but refer the interested reader to Sipser's *Introduction to the Theory of Computation* if any technical questions do arise [126].

Given a combinatorial object, and a property, in many cases there exists a natural translation to a propositional formula, such that the formula is satisfiable *if and only if* the object has the property. Technically, such translations called *reductions* from one decision problem to another [126]. There is a deep technical reason for why such reductions from combinatorial problems to SAT often exist. In 1971, Cook showed that SAT has a property which is now commonly known as NP-completeness [38]. A a consequence of this, all decision problems in the class NP can be reduced to SAT, with only polynomial runtime overhead. In other words, we can use SAT as a general purpose compute engine to solve a large class of decision problems. When we say large, note that NP contains all problems that can be solved in polynomial time on a non-deterministic Turing machine. Equivalently, these are problems whose solutions can be checked in polynomial time. Thus, NP includes problems ranging from primality testing to graph coloring, and from job scheduling to equivalence checking. Indeed, many problems that we are faced with on a daily practice are contained in NP. Figure 1.1 shows representation of some important complexity classes and the relations between them. Famously, it is currently unknown if P = NP. However, due to the time hierarchy theorem we know that $\mathrm{P} \subsetneq \mathrm{EXPTIME}$.

In recent years, there has been a lot of progress in the development of so-called SAT solvers. These are programs which specialize in solving SAT problems efficiently. Due to the NP-complete nature of SAT, fast solvers allow us to solve many problems in NP efficiently. Hence, as these solvers have become more powerful, interest in them has grown accordingly. In the context of *Electronic Design Automation* (EDA), the main application of SAT has traditionally been in hardware verification and other formal tasks.

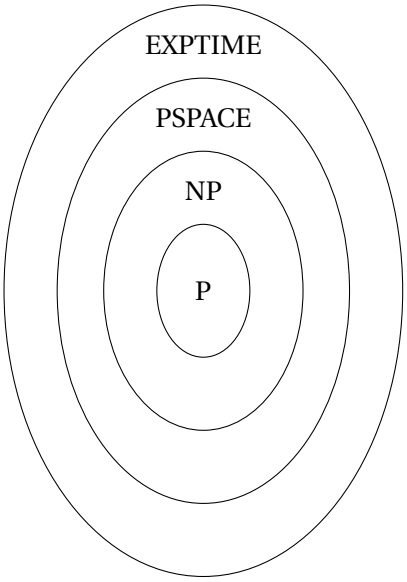


Figure 1.1 – A conceptual representation of some important complexity classes.

In this thesis, we examine the reduction of multi-level logic synthesis to SAT. Indeed, it turns out that there exist a number of natural ways to encode the the synthesis problem as SAT formulae. [1] In fact, logic synthesis can often be viewed as an optimization problem, which is different from a decision problem. However, as we will see, logic synthesis and optimization can be reduced to solving sequences of SAT problems. The research question that this thesis attempts to answer can be summed up as: "*Can SAT be used as an efficient engine for the synthesis of multi-level logic networks, and if so, how?*". We present various SAT encodings of the synthesis problems as well as different techniques for solving it efficiently.

1.1 Boolean Functions

The main object of study in this thesis is the Boolean function. As noted by Ryan O'Donnell in his book *Analysis of Boolean Functions*, Boolean functions are perhaps the most basic objects in computer science [103]. However, their applications range from combinatorics, random graph theory, and statistical physics, to Gaussian geometry, and social choice theory. For these reasons, we take some time to introduce some terminology here.

A completely specified Boolean function f is a mapping between two Boolean spaces. We denote this by $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$. Such functions are commonly known as multiple-output Boolean functions, with n inputs and m outputs. In the context of logic synthesis, $\mathbb{B}$ is usually defined to be $\{0,1\}$. Generally, $\mathbb{B}$ could contain any two distinct objects which obey Huntington's

[1] Note that this may be viewed as comparing the efficiency of different reductions.

postulates for Boolean algebras [88, p. 67]. Because the n-dimensional space $\{0,1\}^n$ can be visualized as the unit cube in n dimensions, it is commonly referred to as the n-dimensional *hypercube*. Points in the n-dimensional Boolean space correspond to the vertices of the cube. Such points may be represented by n-dimensional vectors, which are commonly known as *minterms*. Note that a Boolean function is a map between minterms. There are 2^n minterms, and hence 2^{m2^n} complete Boolean functions from $\mathbb{B}^n$ to $\mathbb{B}^m$. It is sometimes convenient to use a vector notation to represent minterms. We write $(x_1, x_2, \ldots, x_n)$ to an n-dimensional minterm vector.

In logic synthesis it is often the case that, under certain conditions, we do not care what the output of a function is. We call these *don't care* conditions. Such conditions may occur when the value of a sub-circuit cannot be observed at the global circuit outputs, or when certain input patterns are never activated. To support synthesis of such functions, we can extend the definition of Boolean function as a mapping from a Boolean space to an augmented Boolean space which also contains a special don't care element, often represented by $*$. Using this extended definition we denote the mapping by $f : \{0,1\}^n \rightarrow \{0,1,*\}^m$. In other words, to signify that we don't care about a certain minterm, we map it to the $*$ element.

The don't care conditions of a function can be viewed as a set of minterms, or equivalently, as a function. Following the convention of [88], we do not distinguish between either view. Suppose function $f = abx + a'cx$ has a don't care set that is specified by the function $DC = ab'x + a'x'$. We can then use this information to simplify and obtain $f = ax + a'x$. We call the minterms covered by DC the *don't care set*. Conversely, all minterms not covered by DC are part of the *care set*.

Notation. Throughout this thesis, we will use a number of Boolean operators quite frequently. For completeness, we define their notation here in Table 1.1. All of these operators are associative, so we typically do not write brackets. Following Knuth, we use angular bracket notation for the majority operator. We use two different variable naming schemas, depending on which is more convenient in a given context. Variables are denoted by either x's and indexed by numbers (such as x_1, x_2), or by lowercase letters of the alphabet, starting from a (i.e a, b, c).

1.2 Logic Synthesis

Logic synthesis can be summed up as the search for representations of Boolean functions. Typically, this search takes into account certain desirable properties of the representation, such as its size, depth, or estimated energy consumption. EDA algorithms can be partitioned into the three broad categories of architectural synthesis, logic synthesis, and physical design [88, 144]. Thus, logic synthesis is one of the most important steps in most EDA flows, and synthesizing efficient logic representations can significantly affect the final result. This parti-

Operator	Symbol	Example Usage
2-input AND	$\wedge$	$x_1 \wedge x_2$
2-input OR	$\vee$	$x_3 \vee x_4$
2-input EXOR	$\oplus$	$x_5 \oplus x_6$
2-input Less-than	$<$	$x_7 < x_6 \Leftrightarrow \bar{x}_7 \wedge x_6$
Arbitrary binary operator	$\circ$	$x_1 \circ x_2$
3-input MAJORITY	$\langle\cdots\rangle$	$\langle abc \rangle$
5-input MAJORITY	$\langle\cdots\rangle$	$\langle abcde \rangle$
n-input MAJORITY	$\langle\cdots\rangle$	$\langle x_1 x_2 \cdots x_n \rangle$

Table 1.1 – Notation for some commonly used Boolean operators.

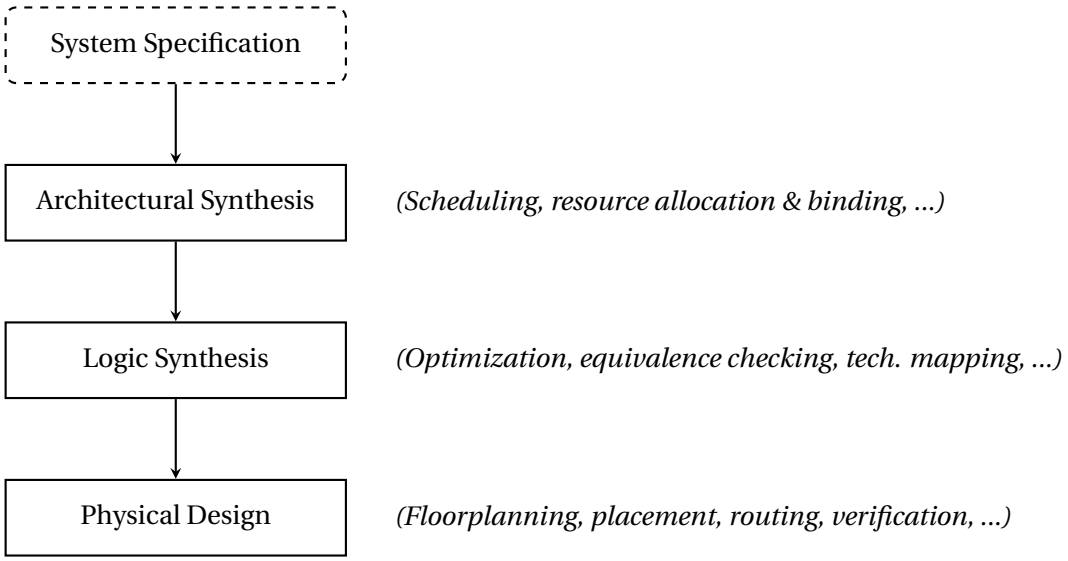


Figure 1.2 – Example of a simplified EDA flow.

tioning of EDA algorithms is an approximation: besides synthesis algorithms, these categories also contain algorithms for tasks such as verification and test. In this thesis we consider only *combinational* logic synthesis, which is the synthesis of circuits without memory elements. Figure 1.2 shows an example of a simplified EDA flow. Typically, hardware designs are specified in some *hardware description language* (HDL). This HDL description then gets compiled into increasingly concrete and low-level representations until finally a physical description is realized, which can then be manufactured.

The number of levels in a logic representation refers to the maximum depth of operators allowed within the representation. This notion of depth can be defined for any algebraic expressions, whether these expressions are Boolean or not. Consider the following two algebraic

expressions:

$$f = a \cdot b + f \cdot g \qquad\qquad g = a(b + c(d + e \cdot f)) \tag{1.1}$$

Expression f applies an addition operator to two product operators. Hence, it has an operator depth of two and is considered a two-level expression. Similarly, expression g is a four-level expression.

Synthesis of logic representations with many levels is known as *multi-level logic synthesis*. The limiting case of multi-level synthesis is a representation consisting of only two logic levels (note that one-level synthesis is trivial). Such representations often have a more simple and regular structure, which lends itself to be exploited by efficient and specialized algorithms. Due to their specialized nature, we commonly distinguish them from the general case as *two-level logic synthesis* algorithms. This thesis is primarily concerned with the use of SAT in multi-level logic synthesis, but for completeness, and to introduce the required terminology, in this section we briefly present both classes.

In addition to the classification into two-level and multi-level logic synthesis, we typically distinguish between exact and heuristic logic synthesis algorithms. Exact algorithms can be used to find optimum logic representations, whereas heuristic algorithms are used to find solutions that are some approximation of the optimum. The use of the term *exact* is due to the historical development of this area in logic synthesis. It can be viewed as referring to the ability to exactly satisfy certain cost criteria in a given logic representation. This will be made more concrete below. Some readers may find the phrase optimal algorithm more familiar or suitable. Optimization in the two-level domain is often referred to as *minimization*.

The typical abstraction in the two-level domain is the *expression form*. There are multiple such forms, corresponding to various types of two-level logic representations. A common one is the *sum of products* (SOP) form. Many readers will have come across this representation before. SOPs can be written as a normal algebraic expressions, although the rules for manipulating them are different, as they must satisfy the axioms of Boolean algebra. Any Boolean function can be expressed as the sum of products of literals, where a literal is a variable or its negation. In SOPs, addition corresponds to the logical disjunction operator $\vee$ and products correspond to the conjunction operator $\wedge$. Thus, we would write the function $f(x_1, x_2, x_3) = x_1 \wedge x_2 \vee \bar{x}_2 \wedge x_3$ as $x_1 \cdot x_1 + \bar{x}_2 \cdot x_3$, or even more succinctly as $x_1 x_2 + \bar{x}_2 x_3$. Expression f in Equation 1.1 is another example of an SOP expression. Other popular two-level representations exist. For example, the *product of sums* (POS) representation is the conceptual dual of the SOP representation. The *exclusive sum of products* (ESOP) representation replaces the disjunctions in the SOP representation by exclusive-OR operators. ESOPS can represent important classes of Boolean functions more compactly than SOPs [117].

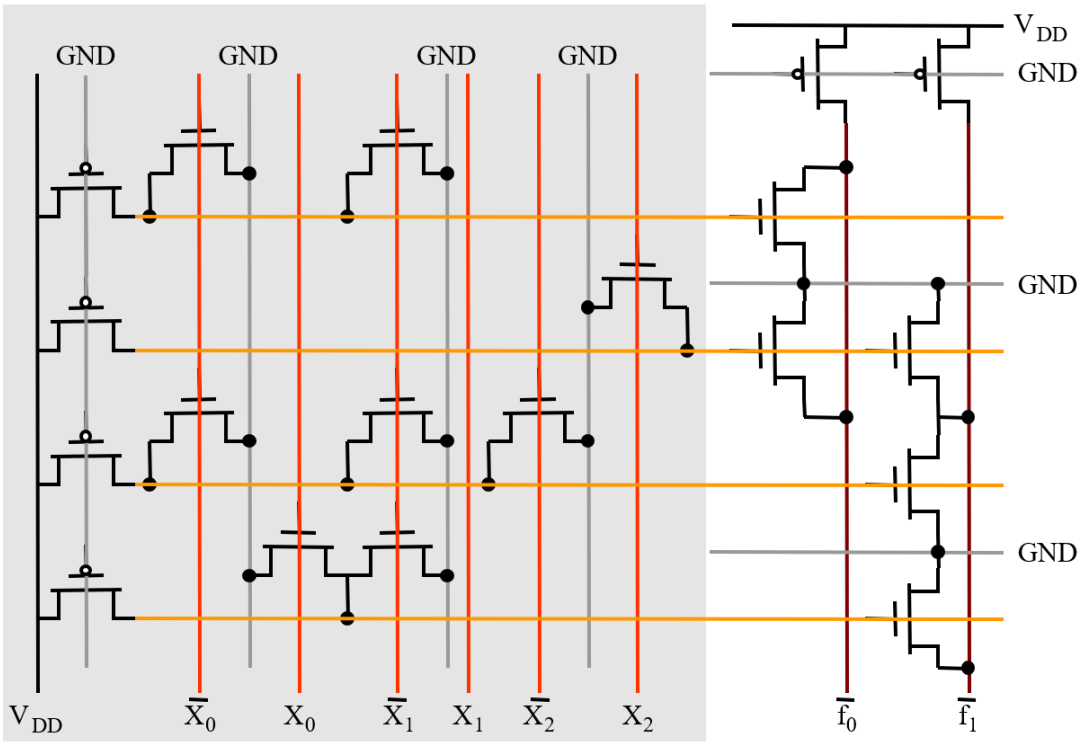


Figure 1.3 – Schematic illustration of a simple NMOS PLA. Courtesy of Professor Jan M. Rabaey.

Historically, two-level logic minimization was born out of a need to optimize *programmable-logic arrays* (PLAs). A PLA is an electronic component which consists of an array of transistors that are aligned in rows. Each row in a PLA can be viewed as a product term, and each column in the array as an input or output variable. See Figure 1.3. Thus, by minimizing the number of product terms and literals in an SOP expression, we reduce the number of rows and transistors needed in the PLA representation of a function. Over the years, many exact and heuristic two-level minimization algorithms have been developed for SOP, POS, and ESOP representations. While the first of these algorithms dates back to the 1950s, some are still used in practice today. Notable examples are the heuristic SOP minimizer `ESPRESSO`, and the ESOP minimizer `EXORCISM`. An overview of these methods is outside of the scope of this thesis, but we refer the interested reader to [108, 85, 24, 106, 118, 136, 137, 91]. As a reference, many of these techniques are also described in [88].

Circuits are often designed as a composition of logic gates. Connecting these gates over multiple levels provides a flexibility that is not achievable by two-level logic representations, allowing for circuits designs with less area and delay. This is in large part due to the sharing of logic that can only be achieved by connections across multiple levels [26]. Logic synthesis

in this paradigm is commonly referred to as multi-level logic synthesis. Although generally beneficial, a drawback of multi-level synthesis is its increased complexity as compared to two-level synthesis.

Whereas in two-level logic the main abstraction is the expression form, in multi-level logic it is the notion of a *logic network*. Multiple models of logic networks exist, just as there are different expression forms. Recent years have seen a rise in the popularity of *homogeneous logic networks* such as *and-inverter graphs* (AIGs) and *majority-inverter graphs* (MIGs) [23, 7]. Such networks consist of a single type of gate. This gate type may be functionally complete; i.e. it may be composed to compute arbitrary Boolean functions. For example, NAND networks are built from a single such operator. AIGs and MIGs, on the other hand, are built from AND and MAJORITY operators, respectively. Since these operators can only implement monotone functions, we typically allow for the use of complemented edges in order to implement inversion. The simplicity of homogeneous networks allows for the creation of efficient data structures and optimization algorithms. This becomes particularly useful as the size and complexity of practical circuits continue to scale. So-called *heterogeneous logic networks*, on the other hand, allow different gate types to be used throughout the network [26, 88]. Common gate types correspond to the (N)AND/(N)OR/INV operators that are well-known from Boolean algebra. In the class of homogeneous logic networks, we can make a further distinction distinguish between *bound* and *unbound* networks. Bound networks allow only a fixed set of Boolean operators to be used as gate types. Unbound logic networks allow gates to implement arbitrary Boolean functions. In that case, local gate functions are often represented by some other common logic representation such as SOPs. Finally, there is a notable class of *canonical* multi-level representations. Canonicity refers to the fact that any two of such representations are equal if and only if the underlying functions they represent are the same. The main example is the *binary decision diagram* (BDD). Originally proposed by Lee and Akers, Bryant extended their work by showing how BDDs could be reduced and manipulated into canonical form, as well as how they could be implemented efficiently. His work resulted in the popularization of BDD-based logic synthesis and verification algorithms [79, 3, 27].

It is common to view multi-level logic networks as *directed acyclic graphs* (DAGs). In this view nodes (vertices) in the DAG correspond to gates in a circuit. Arcs between nodes represent wires between gates. The primary inputs and outputs of the circuit are represented by nodes that have no fanin and no fanout, respectively. Figure 1.4 shows an example. The operators allowed in this example are the set of all 2-input Boolean operators. Note that the vertices are labeled with the Boolean operators corresponding to the gate functions.

As in two-level logic, there exist both exact and heuristic optimization methods for multi-level logic. Exact methods have not been widely used in practice due to their prohibitively high computational complexity [88, p. 343]. In practice, a wide range of heuristic methods

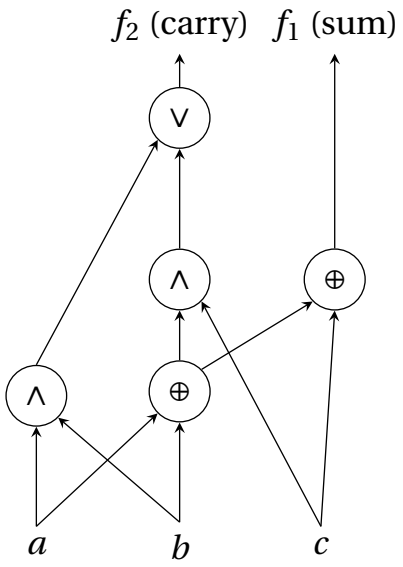


Figure 1.4 – Illustration of a bound heterogeneous Boolean logic network for a full adder.

is applied instead. We briefly mention some of them here. Algebraic methods dispense with the Boolean abstraction and minimize Boolean expressions as though they are ordinary algebraic expressions. This assumption allows these method to be simple and fast. However, they are suboptimal, as they cannot take into account the additional flexibilities allowed by Boolean algebra. As a counterpoint, Boolean methods have been developed, which take into account don't care conditions that are, for example, induced by the structure of the logic network. Boolean methods exploit the properties of Boolean logic more fully than algebraic optimization do. As such, they can be used to reach more optimal solutions, at the expense of some runtime overhead. There is also a wide range of heuristic decomposition methods. These may be applied in throughout a logic network to restructure local sections For detailed descriptions of these methods and more, please refer to [25, 26, 123, 88, 23, 7, 8].

Notwithstanding the classification into two- and multi-level logic, other classes of logic representation do exist and have been studied. For example, there is a notable literature on EXOR based logic minimization using *sum of pseudoproduct* (SPP) expressions [82, 18]. SPPs are three-level logic expressions in which the two-level concept of cubes is generalized to *pseudocubes*, which are products of EXOR factors. A k-SPP is an SPP that has EXORs with k-bounded fanin [31]. Three-level forms such as SPPs and k-SPPs have the advantage that they can represent functions more compactly than two-level forms such as (E)SOPs. Exact and heuristic algorithms to minimize such expressions have been developed. We refer the interested reader to [82, 31, 18].

1.2.1 Exact Synthesis

Exact synthesis is a term used by the logic synthesis community for any method that can be applied to yield *exact* results for logic synthesis problems. In this context, the term exact synthesis is not used in opposition to approximate synthesis, which is a paradigm concerned with the synthesis of systems that produce approximately correct results [87]. Rather, exact synthesis refers to synthesizing logic representations that *exactly meets a specification.* For example, given a Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$ and an number $r \in \mathbb{N}$ we may ask

> $\mathcal{Q}_1$: "Does there exist a logic network N such that N implements f with exactly r gates?"

or

> $\mathcal{Q}_2$: "Does there exist an SOP expression E with exactly r cubes that represents f?"

An exact synthesis algorithm can be used to answer such questions. Typically, we are interested in constructive algorithms. In other words, if a question $\mathcal{Q}_x$ can be answered in the affirmative, we would like to know what is the actual logic representation that meets the specification. In the above examples, we would like our algorithm to produce a logic network N or an SOP expression E.

The notion of exactness is closely related to that of *optimality*, although it is strictly speaking different. Given an algorithm for the exact synthesis of some representation form, we can often adapt it to synthesize optimum representations. Suppose we have a constructive algorithm for $\mathcal{Q}_1$. We could then use it to synthesize size-optimum logic networks as follows. Initialize r to zero and query the algorithm. Increment r until we find the first value r' for which the algorithm reports success. This r' must then be the size of the smallest, i.e. size-optimum, logic network for f. Due to the close correspondence between exact- and optimum synthesis, in logic synthesis literature the terms are often used interchangeably. In practice, the term exact synthesis is widely used to refer to the synthesis of optimum representations.

Exact synthesis algorithms exist for both two- and multi-level logic representations. The Quine-McCluskey algorithm and Petrick's method are well-known algorithms for the minimization of SOPs [108, 85]. Similar methods have been developed for so-called ESOPs as well [117]. In multi-level logic synthesis we encounter various exact minimization algorithms. For instance, Davidson created an algorithm to find the exact minimum NAND decomposition of arbitrary functions [41]. In [77], Lawler generalizes the notion of prime implicant to multi-level logic and develops an exact multi-level optimization algorithm based on that abstraction. In 1962, Roth

and Karp proposed a general-purpose decomposition technique [113], generalizing the earlier work by Ashenhurst [11]. They also showed how this decomposition method may be used as the basis for an search algorithm that find optimum circuits. More recently, enumeration-based techniques have been developed independently by Knuth and Amarù [69, 10]. In practice, heuristic methods are often preferred for performance reasons [44]. The heuristic counterparts to two-level exact synthesis are the `ESPRESSO` and `EXORCISM` algorithms [26, 136].

1.3 Motivation

SAT-based exact synthesis has various practical as well as theoretical applications. Practical applications range from logic optimization, technology mapping, and synthesis for emerging technologies to less obvious ones such as cryptography [96, 53, 131, 132, 133, 139]. All of these can be considered as motivations for this work. In this section, we describe some of the motivations for exact synthesis in general, as well as SAT-based synthesis in particular, in more detail.

A major class of multi-level logic optimization algorithms are known as *logic rewriting* algorithms. We provide a brief description here, and a more detailed one in Section 5.2. In these algorithms, a logic network is restructured by replacing small subnetworks by their optimized counterparts. Typically, the network is partitioned into subnetworks through a process of cut enumeration [37, 81, 98]. A cut of a node n can be defined as a set of nodes c such that any path from the network inputs through n must contain a node in c. Thus, a cut defines a, possibly reconvergent, *logic cone* rooted at n. Cuts implementation may be implemented very efficiently [98], and is therefore a commonly used technique for dissecting out a subnetworks. In rewriting algorithms, optimization of subnetworks is commonly achieved by precomputing a database of highly optimized (or even optimum) networks for some small set of functions, or classes of functions. For example, it is easy to compute and store the 222 NPN classes for all 65,536 single-output 4-input functions [50, 15]. With such a database one can construct a fast optimization algorithm which iterates over the logic network, visiting subnetworks in topological order, iterating over their cuts, and matching them with their optimized versions in the database [93]. Figure 1.5 shows what such a synthesis flow might look like. Global optimization of the logic network $\mathscr{N}$ is achieved by local optimizations of its subnetworks. Note that, in principle, the global optimization flow is independent how local optimization is achieved. Although subnetworks could be retrieved from a precomputed database, they may just as well be computed at runtime by an arbitrary optimization algorithm.

A drawback of conventional logic rewriting algorithms is that only a relatively small number of functions can be stored in a database for retrieval. This means that we are limited to rewriting only small subnetworks, such as subnetworks with up to 4 inputs. This problem is exacerbated when we want to want to rewrite subnetworks with multiple outputs (i.e. windows), or when

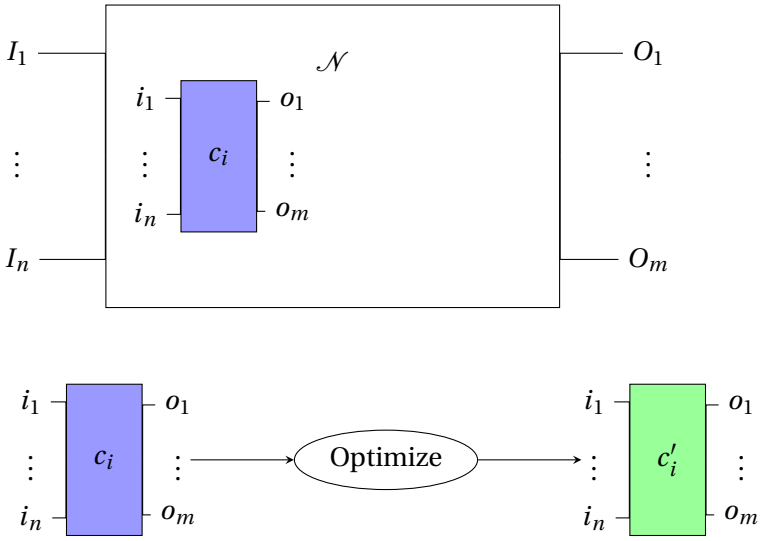


Figure 1.5 – A logic rewriting flow. Optimization of the subnetworks may be achieved in various ways, such as database retrieval, heuristic decomposition, or exact synthesis.

we want to take don't care conditions into account. An efficient exact synthesis algorithm allows us to rewrite larger parts of the network, thus achieving a more global optimization of the overall logic network [53].

Another major driver of exact synthesis is the emergence of novel device technologies, including post-CMOS technologies. Many devices based on emerging nanotechnologies have different behavior than those encountered in CMOS. For instance, they may inherently support different logic primitives than the conventional (N)AND/(N)OR/INV paradigm, such as the majority operator. Examples of are nanoelectromechanical (NEM) relays, spin-wave devices, and quantum-dot cellular automata [78, 120, 80]. Synthesis methods which take full advantage of these primitives can achieve significantly better results [72, 51].

Besides supporting exotic logic primitives, novel technologies may be subject to complex constraints. For example, some emerging technologies do not support inversion in an efficient way, have restrictions on datapath depth, or limited fanout capabilities [141]. In a scenario where solutions must obey such constraints, heuristic algorithms may be too weak. They may find a solution that satisfies the constraints, but they may also fail. Moreover, the failure of a heuristic algorithm to find a solution is no proof of the non-existence of such a solution. The solution may simply be outside the reach of a particular heuristic. In such scenarios, SAT-based exact synthesis compares favorably to both heuristic algorithms as well as other exact synthesis algorithms. First, it can easily be adapted to support exotic logic primitives or complex constraints. This can typically be done by simply adding additional constraints to the SAT formulation or by slight alterations to existing clauses. Second, a SAT formulation can be

used to *prove* whether or not a solution to a specification exists at all.

Exact synthesis also has theoretical applications. For instance, it allows us to derive upper and lower bounds on the combinational complexity of Boolean functions. Kulikov shows how exact synthesis can be linked to such bounds in [76]. Using exact synthesis, Knuth has shown that all 5-variable Boolean functions can be represented using 2-input gate-level networks with at most 12 gates [69, p. 105].

In recent years, significant strides have been made in SAT solving algorithms [19]. These developments, coupled with increases in compute power, have led to a resurgence of algorithms backed by SAT solvers [53, 131, 132]. Despite this progress, the adoption of SAT-based exact synthesis has been limited, due to its unpredictable, and potentially long, runtime. There have been attempts to mitigate runtime with techniques such as the development of alternative CNF encodings, the addition of symmetry breaking clauses, and the use of *counterexample-guided abstraction refinement* (CEGAR) [69, 32]. However, these techniques are often applied in an ad-hoc matter. Moreover, it is not clear how the various encodings and constraints interact with different SAT solvers. To date no comprehensive quantitative comparison of the various methods exists. This presents difficulties in the design of new systems, as there is no data to use as a basis for any design choices. Another hurdle is that, like many EDA algorithms, SAT is difficult to parallelize. Some efforts have been made in parallelizing SAT solvers using techniques such as *cube-and-conquer*, clause sharing, and *portfolio* SAT solvers which apply different SAT solvers in a parallel or distributed manner [57, 62]. This has proven difficult, partially due to theoretical limitations of the resolution procedure [68]. Moreover, solvers based on these methods are typically domain agnostic, and do not take advantage of specific domain structure. Part of the motivation of this thesis is to catalogue and analyze these issues, while also proposing ways to mitigate them.

Thus, we see that there is a broad range of applications for exact synthesis. This, combined with the recent progress made by state-of-the-art SAT solvers, as well as the advantages offered by SAT-based synthesis, forms the motivation for the work in this thesis. In Section 1.4, we provide an overview of the contributions made in this work.

1.4 Thesis Contributions

The contributions made in this thesis can be divided into two main categories:

1. Contributions to the core algorithms for multi-level exact logic synthesis. This includes different encodings and solving strategies.
2. Practical and theoretical applications of exact synthesis. This includes contributions to global logic restructuring algorithms as well as the use of exact synthesis in function

classification.

Here, we summarize these contributions, starting with the core algorithms.

1.4.1 Encodings & Quantitative Comparisons

Although SAT-based exact synthesis is a versatile technique, its runtime behavior may be unpredictable and slow, due to NP-complete nature of SAT. There have been attempts to mitigate runtime through methods such as alternative CNF encodings, symmetry breaking clauses, and the use of *counterexample-guided abstraction refinement* (CEGAR) [69, 32]. However, these techniques are often applied in an ad-hoc matter. Moreover, it is not clear how the various encodings and constraints interact with different SAT solvers. To date no comprehensive quantitative comparison of the various methods exists. Finally, there does not exist a comprehensive review of the various encodings.

The first contribution of this thesis is to present detailed descriptions of the various encodings. Thus, one may use this thesis as a reference for SAT-based exact synthesis algorithms. Moreover, we present a series of experiments which demonstrate, for the first time, quantitative differences between CNF encodings. The descriptions and quantitative results can be used as a basis for the design and implementation of SAT-based exact synthesis systems. The experiments are implemented with the open source *percy* tool, which is available to the public at https://github.com/whaaswijk/percy.

1.4.2 DAG Topology Families

We propose a new type of constraint based on families of DAG topologies. Such families restrict the search space considerably and let us partition the synthesis problem in a natural way. Our approach shows significant reductions in runtime as compared to state-of-the-art implementations, by up to 63%. Moreover, our implementation has significantly fewer timeouts compared to baseline and reference implementations, and reduces this number by up to 61%. In fact, our topology based implementation dominates the others with respect to the number of solved instances: given a runtime bound, it solves at least as many instances as any other implementation. Thus, we show how domain specific knowledge can be used to aid the SAT engine.

1.4.3 Parallel Synthesis

A common drawback of SAT is that the algorithms used by state-of-the-art solvers are hard to parallelize. Of course, this drawback is not limited to SAT. Indeed, it is a common drawback of

logic synthesis algorithms. In this thesis, we show how DAG topologies can be used to inject parallelism into the synthesis problem. We show how topology information can be used to transform the SAT-based exact synthesis problem into an embarrassingly parallel one. This allows us to design parallel algorithms that are up to 68x faster than the state-of-the-art.

1.4.4 Applications

Practical. Some logic rewriting algorithms use exact synthesis to replace small subnetworks by their optimum representations. However, conventional approaches have two major drawbacks. First, their scalability is limited, as Boolean functions are enumerated to precompute their optimum representations. Second, the strategies used to replace subnetworks are not satisfactory. We show how the use of exact synthesis for logic rewriting can be improved. To this end, we propose a novel method that includes various improvements over conventional approaches: (i) we improve the subnetwork selection strategy, (ii) we show how enumeration can be avoided, allowing our method to scale to larger subnetworks, and (iii) we introduce XOR Majority Graphs (XMGs) as compact logic representations that make exact synthesis more efficient. We show a 46% geometric mean reduction (taken over size, depth, and switching activity), a 7% size reduction, and $\mathit{depth} \cdot \mathit{size}$ reductions of 9%, compared to the academic state-of-the-art. Finally, we outperform 3 over 9 of the best known size results for the EPFL benchmark suite, reducing size by up to 12% and depth up to 47%.

Theoretical. One theoretical application of exact synthesis is Boolean function classification. Indeed, it was the method used by Knuth to classify functions in terms of their combinational complexity and minimum depth [69], although he uses enumeration-based exact synthesis as opposed to SAT. We show how a parallel implementation of our SAT-based exact parallelized, which we use to obtain a speedup of approximately 48x. By combining our method with NPN canonization, we find for the first time the minimum-sized logic networks for all 4- and 5-input functions in terms of 3-input Boolean operators.

1.5 Thesis Outline

The remainder of this thesis is organized into five chapters and an appendix. The chapter contents correspond to the main areas in which our contributions fall, as described above.

Chapter 2 - In this chapter we start by describing, in detail, the workings of SAT-based exact synthesis. We provide some background and give pointers to existing literature. We then describe different SAT encodings for exact synthesis, and provide quantitative comparisons between them. Finally, we analyze the computational complexity of SAT-based exact synthesis, and how it relates to the minimum circuit size problem (MCSP).

Chapter 3 - Here, we make the observation that information about the DAG structure of a network can be used to speed up the search for an optimum logic representation. In this chapter, we further describe how such DAG topology information can be used by SAT-based exact synthesis algorithms, and that significant runtime improvements can be obtained by doing so. Additionally, we show how topology information can be used to partition the SAT search space. This leads us to proposing an embarrassingly parallel exact synthesis method, based on families of DAG topologies. We show how our domain-specific parallel method outperforms both single-threaded performance, as well as a state-of-the-art parallel SAT solver.

Chapter 4 - Here begins the second part of this work, in which we look at applications of SAT-based exact synthesis. In this particular chapter, we examine the task of classifying Boolean functions in terms of their intrinsic difficulty. This is of theoretical interest, as helps us understand the distribution of Boolean functions and optimum Boolean chains. It can also help us to find better bounds on circuit sizes. Finally, the techniques we have used to establish these results, such as efficient NPN canonization, can be applied in more practical settings as well, as we show in Chapter 5.

Chapter 5 - One of the most important practical applications of exact synthesis is as a core engine in logic rewriting. In this chapter, we introduce a novel data structure known as XOR-Majority Graphs (XMGs). This compact logic representation is well suited for fast exact synthesis, and for representing XOR/MAJ-heavy logic, such as that occurring in arithmetic units. Combining this new representation, and our exact synthesis algorithms, we introduce a new type of logic rewriting algorithm which does not rely on any sort of precomputation. We show how it can be used to find improvements over state-of-the-art rewriting algorithms.

Chapter 6 - This chapter is an extension and generalization of the work in Chapter 5. We present a generic logic rewriting algorithm for arbitrary k-feasible Boolean networks. As it executes, our algorithm constructs a conflict graph. This graph indicates which subnetworks cannot be rewritten simultaneously. We use it to take both a local and a global view of the optimizations that are achievable by rewriting the network. We try to maximize the possible gain by solving, approximately, the maximum weighted vertex independent set problem on the conflict graph. Finally, we show how our new algorithm finds improvements over state-of-the-art algorithms.

Chapter 7 - In this final chapter, we conclude by summarizing our findings and discussing the contributions of this work. We also describe some open problems, and finish by painting a future outlook.

Appendix A - This appendix describes the motivation behind, and design of, a state-of-the-art exact synthesis library called *percy*. One of the main design goals of *percy* is to allow one to

quickly prototype and experiment with various SAT-based exact synthesis methods. To that end, it implements all synthesis methods described in this thesis. The appendix describes the library design and components in more detail. It also contains several concrete code examples. Although simple, these scripts demonstrate how the *percy* API can be used in practice, and how it interacts with some of the other EPFL logic synthesis libraries.

Core Algorithms Part I

2 Synthesis & Encoding

There are various reductions from the exact synthesis problem to SAT. We commonly refer to such reductions as *encodings*. Encodings are CNF formulae which can be solved to find instances of logic networks that satisfy the specified constraints (or to prove that no such networks exist). Thus, once an encoding has been constructed, we can use the CNF as input to a SAT solver. Given a satisfying solution, we can *decode* the CNF formula to extract a logic network from it. More advanced algorithms can take advantage of a technique known as *counterexample guided abstraction refinement* (CEGAR). In essence, this technique allows us to synthesize logic networks from partial encodings. As such, it can be used to construct faster synthesis algorithms.

In this chapter, we discuss different algorithms for multi-level exact synthesis based on different CNF encodings. We describe these encodings in detail and analyze the trade-offs between them. We start in Section 2.1 by providing some background, relevant definitions, and pointers to existing literature. Then, in Section 2.2, we describe and analyze in detail three different CNF encodings. Following that, in Section 2.3 we discuss CNF symmetry breaking techniques which can be used to reduce the SAT solver's search space. In Section 2.4 we present experiments in the form of quantitative comparisons on different sets of benchmarks. We also discuss the impact of the various symmetry breaks. Next, in Section 2.5, we describe how more advanced algorithms can take advantage of CEGAR to improve efficiency. In Section 2.6, we present an extension of the previous algorithms for synthesis with don't cares. Section 2.7 analyzes the computational complexity of exact synthesis. Finally, we summarize our findings in Section 2.8.

2.1 Background

In this section we describe the background necessary to place the rest of this work in context. This includes definitions, notation, as well as terminology used to describe different techniques used throughout SAT-based exact synthesis. The concepts we discuss here will be used extensively throughout the text.

2.1.1 Boolean Chains

We present here our definition of Boolean chains, a concept originally introduced by Knuth [69]. Boolean chains may be viewed as a precise formal model of the concept of multi-level logic networks [44]. As these objects are the main target of our synthesis algorithms, such a precise definition is warranted. Knuth's original formalization is limited to chains consisting of 2-input operators. Here, we extend this definition to k-input operators, where k is arbitrarily large, but fixed.

Essentially, a Boolean chain is a DAG in which every internal vertex has a corresponding k-input Boolean operator $\phi : \mathbb{B}^k \to \mathbb{B}$. Following the convention of Roth and Karp [113], we denote the set of allowed operators by $\mathscr{B}$. Let $f = (f_1, \ldots f_m)$ be a multiple-output Boolean function, such that $f : \mathbb{B}^n \to \mathbb{B}^m$ and the functions $f_1, \ldots f_m$ are defined over common support $x_1, \ldots, x_n$. Then, for $k \geq 1$ and a set $\mathscr{B}$, a k-input operator Boolean chain is a sequence $(x_{n+1}, \ldots, x_{n+r})$, where

$$x_i = \phi_i(x_{j(i,1)}, \ldots, x_{j(i,k)}) \qquad \text{for } n+1 \leq i \leq n+r$$

such that $\phi_i \in \mathscr{B}$, $1 \leq j(i,\cdot) < i$, and for all $1 \leq k \leq m$, either $f_k(x_1, \ldots, x_n) = x_{l(k)}$ or $f_k(x_1, \ldots, x_n) = \bar{x}_{l(k)}$, where $0 \leq l(k) \leq n+r$, and $x_0 = 0$ the constant zero input. For example, in Knuth's definition of Boolean chains, $\mathscr{B}$ is the set of all binary operators. The operators in $\mathscr{B}$ are often referred to as the permissible *logic primitives*, or simply *primitives*. The objects $x_{n+1}, \ldots, x_{n+r}$ are called the *steps* of the chain. For brevity, we occasionally refer to Boolean chains simply as chains.

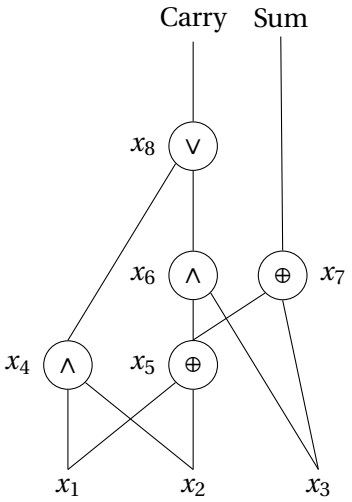


Figure 2.1 – Illustration of an optimum Boolean chain for a full adder.

For example, when $n = 3$, then the 2-input operator 5-step chain

$$\begin{aligned} x_4 &= x_1 \wedge x_2 \\ x_5 &= x_1 \oplus x_2 \\ x_6 &= x_3 \wedge x_5 \\ x_7 &= x_3 \oplus x_5 \\ x_8 &= x_4 \vee x_6 \\ l(1) &= 7 \\ l(2) &= 8 \end{aligned}$$

can be used to represent the 3-input 2-output function $f(x_1, x_2, x_3) = (x_1 \oplus x_2 \oplus x_3, \langle x_1, x_2, x_3 \rangle)$, which is commonly known as a full adder. Figure 2.1 illustrates this example.

The extension of Boolean chains to arbitrary k-input operators has several motivations. First, synthesis of chains with larger operator sizes may be significantly faster. For example, using 3-input operator Boolean chains, one can efficiently classify the set of all 5-input functions using SAT-based exact synthesis [54], whereas this has not been achieved for 2-input operator chains. Second, one application of exact synthesis is in technology mapping, where we are often required to use a diverse set of logic *primitives*. Generally, we cannot assume that a given cell library contains only 2-input operators. Finally, recently there has been a resurgence of bounded logic network representations such as MIGs and XMGs [131]. These require operators ranging from 3 to at least 5, although this depends on the specific representation (i.e. we typically understand MIGs to require 3-input operators).

We say that a Boolean chain is *normalized* or *normal* if all of its steps correspond to normal functions, i.e. functions that output zero when all of their k inputs are zero [69]. For example, a chain consisting of AND and OR operators is normal, but a chain of NANDs is not. As we describe below, the use of normal chains can be advantageous in SAT-based exact synthesis: when a chain is normal, we may not have to encode the first truth table bit of its operators.

It will be useful to define some operators on Boolean chains. We often want to refer to the size of a Boolean chain. Therefore, let the σ operator return the number of steps in a Boolean chain. In other words, given a chain $c = (x_{n+1}, \ldots, x_{n+r})$, we have $\sigma(c) = r$. For example, let c be the Boolean chain illustrated in Figure 2.1. Then, $\sigma(c) = 5$. Next, it is often useful to refer to the function implemented by a chain. We will use the F operator to do so. Thus, $F(c) = f$, where $f = (x_1 \oplus x_2 \oplus x_3, \langle x_1 x_2 x_3 \rangle)$. We sometimes refer to $F(c)$ as the *chain function* of c. Finally, let ω refer to the set of Boolean operators that correspond to steps in a chain. In other words, if $\omega(d) = B$, then all steps in d have a corresponding operator in B. Note that $B \subseteq \mathscr{B}$. We have $\omega(c) = \{\vee, \wedge, \oplus\}$.

A common operation on Boolean chains is that of *simulation*. It is a technique used to (partially) reconstruct the function represented by a particular chain. Given a Boolean chain c on n variables, we need to determine what the output values of the chain are for some subset S of the 2^n minterms. This can be done as follows. Select a minterm $s \in S$, and fix the inputs of the chain to the values specified by the minterm. Then, simply propagate values up the chain by evaluating the chain operators in topological order. Once all operators have been evaluated, the chain outputs represent the chain's function value at that particular minterm. Consider, for example, the chain illustrated by Figure 2.1. Simulating this chain on the minterm $(x1, x2, x3) = (0, 1, 0)$ results in outputs $(\text{Sum}, \text{Carry}) = (1, 0)$, whereas simulating it on $(1, 1, 0)$ results in outputs $(0, 1)$. We can completely reconstruct $F(c)$ by simulating c on all minterms. However, this takes time exponential in n. Let t be a minterm. With a slight abuse of notation we then write $c(t) = b$ to indicate that simulating the chain c on this minterm results in the value b. Note that b is a vector of output values in the general multiple-output case.

2.1.2 SAT-based Exact Synthesis

The first instance of SAT-based exact synthesis that we are aware of is the 2007 tutorial on "Practical SAT" given by Eén at the FMCAD conference [45]. Later, Kojevnikov, Kulikov, and Yaroslavtsev used an extended CNF encoding to find circuit-size upper bounds [71]. Subsequently, Knuth formalized his own encoding, which was limited to 2-input operator chains [70, p. 278]. These algorithms all aim to find size-optimum Boolean chains. Soeken et al. extended them to synthesize depth-optimum chains [132]. In this thesis, our focus is on methods for size-optimum synthesis, which we will often refer to simply as (SAT-based) exact synthesis. However, note that, due to the large overlap in methodology, many the results should carry

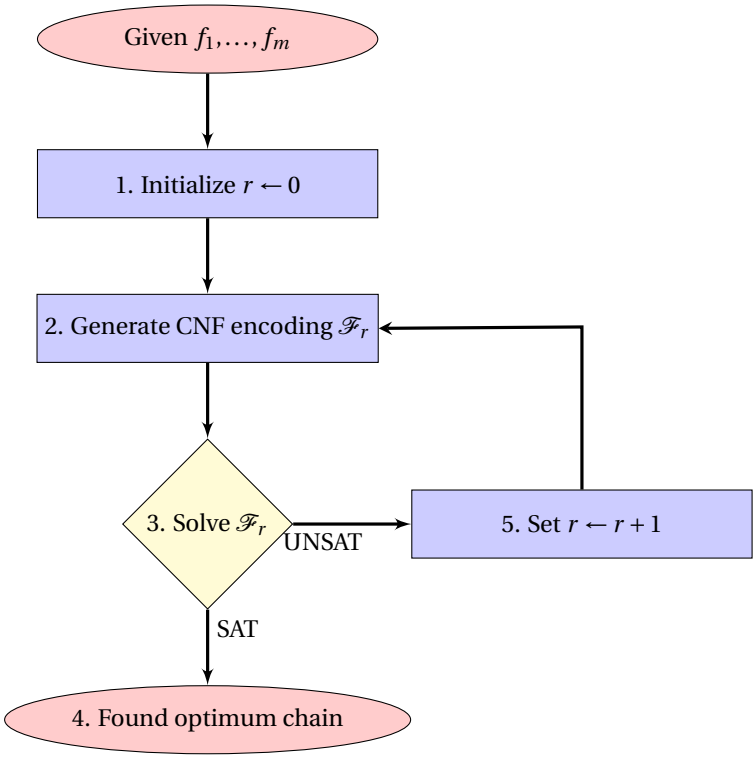


Figure 2.2 – Illustration of a basic size-optimum SAT-based exact synthesis algorithm.

over to the depth-optimum exact synthesis, as well as other forms of SAT-based synthesis with complex constraints.

To formally describe SAT-based exact synthesis, we assume the generic synthesis problem, in which we are given a multiple-output Boolean function $f = (f_1, \ldots, f_m) : \mathbb{B}^n \to \mathbb{B}^m$. Given the tuple $(f, \mathscr{B}, r)$, the exact synthesis decision problem is then defined precisely as the question $\mathscr{Q}_r$:

"Does there exist a Boolean chain c such that $\sigma(c) = r$, $F(c) = f$, and $\omega(c) \subseteq \mathscr{B}$?"

As discussed briefly in Section 1.2.1, the principles behind different methods for SAT-based size-optimum exact synthesis are the same and roughly correspond to the phases illustrated in Figure 2.2. From the different phases of synthesis, we can see that the size-optimum problem can be solved by a sequence of SAT formulae, where each formula $\mathscr{F}_r$ corresponds to the number of steps r. If phase 5 is reached, then we have a proof that no chain with r or fewer steps can implement the function f. Heuristic methods typically cannot give such guarantees [141, 133].

Phase 2 of the synthesis process is made possible by the NP-complete nature of SAT. Different ways of implementing this phase correspond to different encodings of the exact synthesis problem into SAT. We are free to choose between distinct (but equivalent) CNF encodings $\mathscr{F}_r$. However, it may not be clear a priori which one is best in any given context.

The precise variables and clauses used to construct $\mathscr{F}_r$ depend on a number of factors. For example, they may depend on the logic primitives in $\mathscr{B}$. In the encoding introduced by Knuth, all 2-input operators are allowed. However, in the context of synthesis for MIGs or XMGs, we need only encode Majority and EXOR operators. Other factors, such as complex constraints on network structure, or topological restrictions are also important for determining $\mathscr{F}_r$. We discuss the essential CNF clauses necessary for Boolean chain synthesis in Section 2.2. Then, in Section 2.3, we show how additional constraints can be added to avoid symmetries in the SAT search space. Later, in Chapter 3, we show how constraints based on topology families can be advantageous.

2.1.3 A Note on Optimality

It seems prudent here to address a point of confusion which sometimes arises when discussing exact synthesis, and specifically synthesis of optimum logic networks. We always refer to optimality within the context of a specific model of computation. The model of computation used throughout this paper is that of Boolean chains. Suppose we synthesize a function f and obtain the chain c. When we say that c is size-optimum, this means that there exists no chain c' that computes f with fewer steps than c. That is not to say that there may not exist different models of computation, such as cyclic combinational circuits [110], in which f could be implemented with fewer computational primitives.

2.2 CNF Encodings

In this section, specifically Sections 2.2.1 to 2.2.3, we describe three different CNF encodings. This is not meant to be an exhaustive list: other encodings exist, including the one proposed by Kojevnikov et al. [71]. Rather, we present these encodings as they are heavily used in practice, and yet we are unaware of any detailed descriptions or comparisons in existing literature.

In the following, we assume the generic synthesis problem in which we are given the multiple-output Boolean function $f = (f_1, \ldots, f_m) : \mathbb{B}^n \to \mathbb{B}^m$ and $\mathscr{B}$ consists of the entire set of 2-input operators. While all three encodings we describe can be used for the synthesis of k-input operator chains, for clarity we describe only the 2-input case. The extension to arbitrary k is then straightforward.

In other contexts, exact synthesis can typically be framed as a special case of the encodings

we describe here. For example, all of the encodings we describe can be extended and used for synthesis under complex constraints (e.g. fanout restrictions). In cases where only a restricted set of k-input operators are allowed, such as synthesis for majority graphs, MIGs, or XMGs, we can often do so by simplifying or slightly modifying the constraints described in this section [53].

2.2.1 Single Selection Variable (SSV) Encoding

The SSV encoding is typically used for the synthesis of *normal* 2-input operator chains. The normalization requirement does not limit the optimality of synthesized chains: any function computed by a non-normalized chain can be computed by a normalized chain with the same number of steps. One can simply complement the desired non-normal function, synthesize a normal chain, and invert it. The use of normal chains has the advantage that they can be built out of normal steps. This reduces the number of variables needed by the encoding.

In this encoding, $\mathscr{F}_r$ consists of the following variables, for $1 \leq h \leq m$, $n < i \leq n + r$, and $0 < t < 2^n$:

$$
\begin{aligned}
x_{it} &: \text{t}^{\text{th}} \text{ bit of } x_i\text{'s truth table} \\
g_{hi} &: f_h(x_1, \ldots, x_n) = x_i \\
s_{ijk} &: x_i = x_j \circ_i x_k \text{ for } 1 \leq j < k < i \\
f_{ipq} &: p \circ_i q \text{ for } 0 \leq p, q \leq 1, p + q > 0
\end{aligned}
$$

The x_{it} variables capture the global truth table values computed by steps in the chain. They contain the function computed by a specific step, in terms of the chain's primary inputs. In other words, they specify, for each minterm of the function, what the value is computed at a given step. Consequently, these variables are sometimes referred to as *simulation variables* The g_{hi} variables determine which outputs point to which step, and the s_{ijk} variables determine the inputs j and k, for each step i. These are also known as *selection variables*. The f_{ipq} encode for all steps i what the corresponding Boolean operator is. We do not encode f_{i00}, since $f_i(0,0) = 0$ by definition of normal chains. Recall from Table 1.1 that we use $\circ_i$ to denote arbitrary binary operators.

> ***Example.*** We will use the the full-adder in Figure 2.1 as a simple running example throughout this section. The table below shows the simulation variable values corresponding to each step in that particular chain.

t	x_{1t}	x_{2t}	x_{3t}	x_{4t}	x_{5t}	x_{6t}	x_{7t}	x_{8t}
1	1	0	0	0	1	0	1	0
2	0	1	0	0	1	0	1	0
3	1	1	0	1	0	0	0	1
4	0	0	1	0	0	0	1	0
5	1	0	1	0	1	1	0	1
6	0	1	1	0	1	1	0	1
7	1	1	1	1	0	0	1	1

Note that we only need seven simulation variables per step, since we only synthesize normal functions. Furthermore, the truth table values of the primary inputs are constants, and therefore do not need to be explicitly encoded. For clarity we show their virtual values here, but note that they are not included in the actual CNF encoding. In this example, the chain is already synthesized, and its structure (fanin connections) and functionality (step functions) are known. From this we can derive the values of these simulation variables. In other words, once we fix a logic network's structure we can uniquely determine its simulation vectors.[1] In general, of course, the structure and functionality of the chain are not fixed and the value of the simulation variables are undetermined.

In this example, there are ten g_{hi} variables, since there are two outputs and each output may potentially point to fives steps. Exactly two of these output variables which are set to one, indicating which steps correspond to outputs: $g_{17} = g_{28} = 1$. All other g_{hi} are set to zero.

Similarly, from the DAG structure of the network, we can see that $s_{412} = 1$, $s_{512} = 1$, $s_{635} = 1$, $s_{735} = 1$, and $s_{846} = 1$. All other s_{ijk} are zero.

Finally, the variables encoding the Boolean operators are assigned the following values:

$$\begin{aligned} (p,q) &= (1,1)\ (0,1)\ (1,0) \\ f_{4pq} &= 1 \quad 0 \quad 0 \\ f_{5pq} &= 0 \quad 1 \quad 1 \\ f_{6pq} &= 1 \quad 0 \quad 0 \\ f_{7pq} &= 0 \quad 1 \quad 1 \\ f_{8pq} &= 1 \quad 1 \quad 1 \end{aligned}$$

[1] Note that this implication does not work in the other direction.

The full adder can be extracted simply by inspecting the values of the selection and operator variables.

The SSV variables defined above must be constrained by a set of clauses which ensures that the chain computes the correct functions. For $0 \leq a, b, c \leq 1$ and $1 \leq j < k < i$, the main clauses are:

$$(\bar{s}_{ijk} \vee (x_{it} \oplus a) \vee (x_{jt} \oplus b) \vee (x_{kt} \oplus c) \vee (f_{ibc} \oplus \bar{a}))$$

Intuitively, these clauses encode the following constraint: if step i has inputs j and k *and* the t$^{\text{th}}$ bit of x_i is a *and* the t$^{\text{th}}$ bit of x_j is b *and* the t$^{\text{th}}$ bit of x_k is c, *then* it must be the case that $b \circ_i c = a$. This can be understood by rewriting the formula as follows:

$$((s_{ijk} \wedge (x_{it} \oplus \bar{a}) \wedge (x_{jt} \oplus \bar{b}) \wedge (x_{kt} \oplus \bar{c})) \rightarrow (f_{ibc} \oplus \bar{a}))$$

Note that a, b, and c are constants which are used to set the proper variable polarities.

Let $(b_1, \ldots, b_n)_2$ be the binary encoding of truth table index t. In order to fix the proper output values, we add the clauses $(\bar{g}_{hi} \vee \bar{x}_{it})$ or $(\bar{g}_{hi} \vee x_{it})$ depending on the value $f_h(b_1, \ldots, b_n)$. In other words, we find the value of function h for each minterm. Using this value we can determine the polarity of the x_{it} variables: if $f_h(b_1, \ldots, b_n) = 0$ and output h points to step i, then x_{it} must be zero. Otherwise the wrong function would be computed by that output at index t. The case where $f_h(b_1, \ldots, b_n) = 1$ is analogous. We also add $\bigvee_{i=n+1}^{n+r} g_{hi}$ and $\bigvee_{k=1}^{i-1} \bigvee_{j=1}^{k-1} s_{ijk}$, so that every output points to a step in the chain and to ensure that every step has two inputs.

Example. Recalling our full adder example, let $f_1(x_1, x_2, x_3) = x_1 \oplus x_2 \oplus x_3$) be the Sum output. For truth table index $t = 7$, we have binary encoding $(1, 1, 1)$. Hence, since $f_1(1, 1, 1) = 1$, we add the clause $(\bar{g}_{1i} \vee x_{i7})$ for all i $(n < i \leq n + r)$. Similarly, when $t = 5$, we have binary encoding $(1, 0, 1)$, and $f_1(1, 0, 1) = 0$. Hence, we add $(\bar{g}_{1i} \vee \bar{x}_{i5})$ for all i.

A key difference between the encodings in this section is in the number of s_i variables, also known as the *selection variables*, that they use. Let us therefore compute the number of selection variables in the SSV encoding. All possible operand pairs for step i are explicitly encoded by separate variables $s_{ijk} (j < k < i)$. For a given i there are $\binom{i-1}{2}$ possible operand pairs to choose from. Thus, the total number of selection variables in the SSV encoding is

$$\sum_{i=n+1}^{n+r} \binom{i-1}{2} = \frac{1}{6}(3n^2 + 3n(r-2) + r^2 - 3r + 2).$$

In other words, it is quadratic in the number of inputs n and gates r.

2.2.2 Multiple Selection Variables (MSV) Encoding

In the MSV encoding, we define the following variables $1 \leq h \leq m$, $n < i \leq n + r$, and $0 < t < 2^n$:

$$\begin{aligned} x_{it} &: \text{t}^{\text{th}} \text{ bit of } x_i\text{'s truth table} \\ g_{hi} &: f_h(x_1, \ldots, x_n) = x_i \\ s_{ij} &: x_i \text{ has operand } j \text{ where } 1 \leq j < i \\ f_{ipq} &: p \circ_i q \text{ for } 0 \leq p, q \leq 1, p + q > 0 \end{aligned}$$

The MSV encoding uses the variable s_{ij} to indicate that step i has operand j. Thus, it requires only $i - 1$ selection variables per step. The total number is

$$\sum_{i=n+1}^{n+r} (i-1) = \frac{1}{2}(2n + r - 1).$$

Thus, the MSV encoding reduces the number of variables from a quadratic to a linear number, as compared to the SSV encoding. However, it achieves this reduction in variables at the cost of additional clauses. It must maintain the cardinality constraint that $\sum_{j=1}^{i-1} s_{ij} = 2$. In this case, that constraint can no longer be enforced by a single clause. One solution is to add the clauses

$$\bigwedge_{j<k<l<i} (\bar{s}_{ij} \vee \bar{s}_{ik} \vee \bar{s}_{il})$$

and

$$\bigwedge_{k=1}^{i-1} (s_{i1} \vee \ldots s_{i(k-1)} \vee s_{i(k+1)} \vee \cdots \vee s_{i(i-1)}).$$

Intuitively, such clauses work as follows. They state that in any triplet of potential operands for step i at least one must be false. Moreover, consider a set of operands which consists of all potential operands of i with one removed. In such a set at least one operand must be used by i. Thus, by adding this second set of clauses we ensure that at least 2 operands are used. Combined, these constraints therefore ensure that exactly 2 operands are selected. The drawback of these constraints is that they require

$$\sum_{i=n+1}^{n+r} \binom{i-1}{3} + \binom{i-1}{i-2}$$

additional clauses, which is quadratic in n and r.

Fortunately there exist more efficient encoding schemes. One example is to add a *unary binary counter* (UBC) circuit to the CNF. Essentially such a circuit acts as a (partial) ripply-carry adder

which allows us to ensure that the total number of selected operands is equal to 2. Moreover, it uses only a linear number of clauses. Finally, it has the advantage that as soon as 2 operands are selected, the entire circuit is computed by unit propagation, exploiting the SAT solver's efficiency. A complete description of this circuit is outside the scope of this paper, but we refer the interested reader to [125]. We use the UBC encoding in all our experiments.

After putting the appropriate cardinality constraints in place, for $0 \leq a, b, c \leq 1$ and $1 \leq j < k < i$, the main clauses are now:

$$(\bar{s}_{ij} \vee \bar{s}_{ik} \vee (x_{it} \oplus a) \vee (x_{jt} \oplus b) \vee (x_{kt} \oplus c) \vee (f_{ibc} \oplus \bar{a}))$$

Similar to the SSV encoding, we add the clauses $(\bar{g}_{hi} \vee \bar{x}_{it})$ or $(\bar{g}_{hi} \vee x_{it})$ depending on the value $f_h(t_1, \ldots, t_n)$. We also add $\bigvee_{i=n+1}^{n+r} g_{hi}$.

> ***Example.*** Let us consider again the previous example of encoding the full-adder. The MSV encoding is similar to the SSV encoding, with the main difference being in the selection variables. We now have
>
> $$s_{41} = s_{42} = 1$$
> $$s_{51} = s_{52} = 1$$
> $$s_{63} = s_{65} = 1$$
> $$s_{73} = s_{75} = 1$$
> $$s_{84} = s_{76} = 1$$
>
> and all other s_{ij} zero.

2.2.3 Distinct Input Truth Tables (DITT) Encoding

The DITT encoding possesses some interesting structural differences from the previous two. In the SSV and MSV encodings there is a tight coupling between the selection variables and the propagation of truth table bits through the operator variables. The DITT encoding removes that direct coupling at the cost of introducing additional variables and clauses. However, while it creates more variables, it simultaneously reduces the complexity of the clauses.

Let us begin by defining the variables:

$$
\begin{aligned}
x_{it} &: t^{\text{th}} \text{ bit of } x_i\text{'s truth table} \\
x_{it}^{(k)} &: t^{\text{th}} \text{ bit of } x_i\text{'s } k^{\text{th}} \text{ input truth table, } k \in \{1,2\} \\
g_{hi} &: f_h(x_1,\ldots,x_n) = x_i \\
s_{ij}^{(k)} &: \text{Input } k \text{ of } x_i \text{ has operand } j \text{ for } 1 \leq j < i,\ k \in \{1,2\} \\
f_{ipq} &: p \circ_i q \text{ for } 0 \leq p,q \leq 1, p+q > 0
\end{aligned}
$$

The output and operator variables are equivalent to those in the previous encodings. The difference lies in the selection variables and propagation of truth table bits. Previously, we defined t truth table bit variables for each step. In this case we define the additional variables $x_{it}^{(k)}$ which correspond to the truth tables of the inputs to step i. The actual values of those bits depend on which inputs i has selected. In this encoding, we define selection variables for each fanin of a step. Variables for the different fanins are indexed by k, whose range depends on the operator size (2 in this case). Obviously this encoding requires more variables. For example, it encodes three times as many truth table bits. However, it recovers this complexity by reducing the complexity of constraints.

The main clauses are now:

$$((x_{it} \oplus a) \vee (x_{it}^{(1)} \oplus b) \vee (x_{it}^{(2)} \oplus c) \vee (f_{ibc} \oplus \bar{a}))$$

Note the structural difference with the above encodings here. In those, the main clauses combine the selection variables and the truth table bits to propagate truth table and operator bits. The DITT essentially removes this coupling. Instead, the structure-based propagation of truth table bits is determined by adding the clauses

$$s_{ij}^{(k)} \rightarrow (x_{it}^{(k)} = x_{jt}).$$

In other words, the input truth table bits (used in the main clause) are now determined directly by the selection variables.

Finally, we ensure that all step fanins point to some input by adding $\bigwedge_{k=1}^{2} \bigvee_{j=1}^{i-1} s_{ij}^{(k)}$.

Let us count the number of selection variables used in this encoding. Consider a step x_i. Each of its k fanins may select any of the previous $i-1$ steps. Therefore, the number of selection

variables per step is $k(i-1)$. The total number of selection variables for all steps is then

$$\sum_{n+1}^{n+r} k(i-1) = k\sum_{n+1}^{n+r}(i-1) = \frac{k}{2}(2n+r-1).$$

Thus, we require k times as many selection variables as in the MSV encoding. However, the number is still linear in n and r.

There is another subtle difference between this encoding and the previous two. In fact, the DITT encoding is more general. It allows step fanins to be ordered arbitrarily: the k-th fanin of step i may point to step $i'+m$ ($m>0$), even when fanin $k+1$ points to step i'. This flexibility allows it to synthesize a larger class of logic networks as compared to the previous encodings. Those only synthesize Boolean chains which can be viewed of a logic network in which gate fanins are are ordered tuples. Although this flexibility may be desirable in some cases, it also increases the search space. Therefore, in the context of synthesis for Boolean chains we add the additional clauses $\bigwedge_{j=1}^{i-2}\bigwedge_{j'=1}^{j}(\bar{s}_{ij}^{(1)} \vee \bar{s}_{ij'}^{(2)})$ to ensure that all step fanins are ordered.

Example. We again consider the full-adder chain, as we have for the previous two encodings. Let us consider the assignments to the truth variables. We have

$$\begin{aligned} s_{41}^{(1)} &= s_{42}^{(2)} = 1 \\ s_{51}^{(1)} &= s_{52}^{(2)} = 1 \\ s_{63}^{(1)} &= s_{65}^{(2)} = 1 \\ s_{73}^{(1)} &= s_{75}^{(2)} = 1 \\ s_{84}^{(1)} &= s_{76}^{(2)} = 1 \end{aligned}$$

with all other $s_{ij}^{(k)}$ set to zero. Next, let us consider the assignments to the (input) truth table bits:

t	$x_{4t}^{(1)}$	$x_{4t}^{(2)}$	x_{4t}	$x_{5t}^{(1)}$	$x_{5t}^{(2)}$	x_{5t}	$x_{6t}^{(1)}$	$x_{6t}^{(2)}$	x_{6t}	$x_{7t}^{(1)}$	$x_{7t}^{(2)}$	x_{7t}	$x_{8t}^{(1)}$	$x_{8t}^{(2)}$	x_{8t}
1	1	0	0	1	0	1	0	1	0	0	1	1	0	0	0
2	0	1	0	0	1	1	0	1	0	0	1	1	0	0	0
3	1	1	1	1	1	0	0	0	0	0	0	0	1	0	1
4	0	0	0	0	0	0	1	0	0	1	0	1	0	0	0
5	1	0	0	1	0	1	1	1	1	1	1	0	0	1	1
6	0	1	0	0	1	1	1	1	1	1	1	0	0	1	1
7	1	1	1	1	1	0	1	0	0	1	0	1	1	0	1

From these values we can clearly see how the simulation variables x_{it} are computed as a function of the input truth tables and step operators. We also see that the DITT encoding requires significantly more variables to represent the input

truth tables. In fact, it requires an additional $kr(2^n-1)$ variables: for each step we require a truth table vector one for every input. As we will see in the experiments below, these variables do not necessarily make this encoding much slower. Their values are fully implied by the selection and step operator variables. As such, they can be determined by unit propagation, which is a relatively fast operation.

2.3 Symmetry Breaking

The encodings as we have described them so far are sufficient to synthesize any Boolean chain. Here, we describe several optional *symmetry breaking* clauses. These clauses are not required to produce correct results, but they may be used to constrain the SAT solver's search space while still providing exact results. As such, the aim of adding these clauses is to reduce runtime at the cost of additional clauses and CNF encoding complexity. Due to this additional cost, it is a priori not clear how they affect synthesis runtime. In Section 2.4 we present a number of experiments to elucidate their impact. The constraints presented here are due to Kojevnikov et al. [71] and Knuth [70]. We describe them here using the SSV encoding for 2-input chains, but it is straightforward to extend these descriptions to other encodings and input sizes.

Only non-trivial operands (N)

Any optimum Boolean chain will not contain any trivial Boolean operands such as variable projections or the constant 1 and 0 functions. We may exclude these by adding the additional clauses $(f_{i01} \vee f_{i10} \vee f_{i11})$, $(f_{i01} \vee \bar{f}_{i10} \vee \bar{f}_{i11})$, and $(\bar{f}_{i01} \vee f_{i10} \vee \bar{f}_{i11})$. For example, the first of these clauses requires at least one of f_{i01}, f_{i10}, and f_{i11} to be true. In doing so, it disallows steps to use the constant zero operator. Similarly, the second and first clauses disallow variable projections.

Use all steps (A)

An optimum chain must use all its steps to compute its output value; otherwise we could remove the unused steps. To enforce this constraint, we can add the clauses

$$\left(\bigvee_{k=1}^{m} g_{ki} \vee \bigvee_{i'=i+1}^{n+r} \bigvee_{j=1}^{i-1} s_{i'ji} \vee \bigvee_{i'=i+1}^{n+r} \bigvee_{j=i+1}^{i'-1} s_{i'ij} \right)$$

for all i. An example of this symmetry break is given in Figure 2.3.

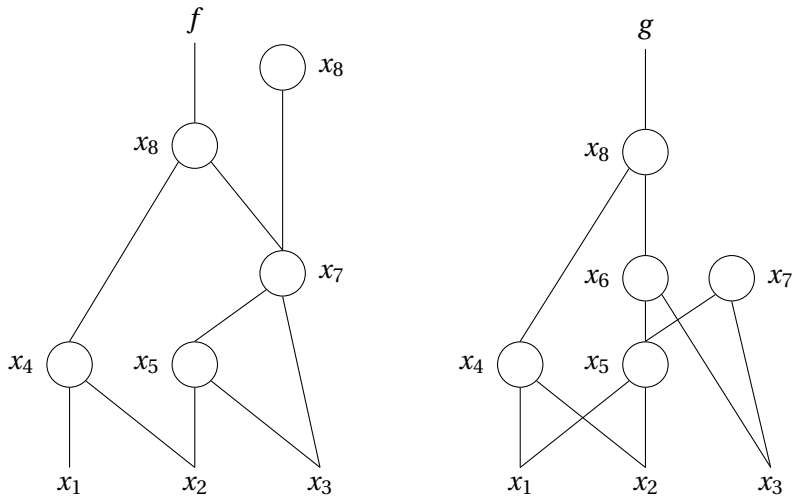


Figure 2.3 – Examples of circuit topologies that are avoided by applying symmetry break **(A)**.

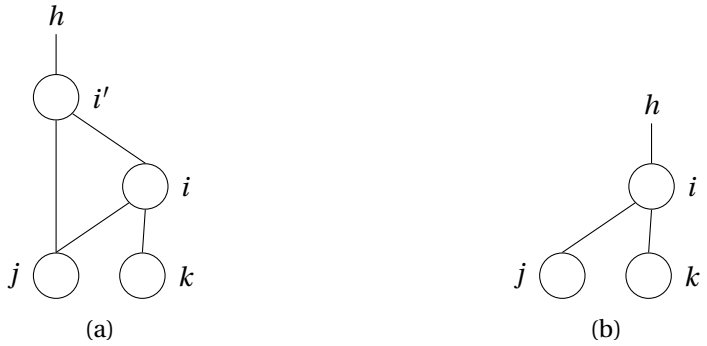


Figure 2.4 – Illustration of the kinds of circuit structures avoided by symmetry break **(R)**.

No re-application of operands (R)

Adding the clauses $(\bar{s}_{ijk} \vee \bar{s}_{i'ji})$ and $(\bar{s}_{ijk} \vee \bar{s}_{i'ki})$ for $i < i' \leq n + r$ ensures that the chain never *re-applies* an operator. Intuitively, suppose that step i has inputs j and k. If $i' > i$ has inputs j and i (or k and i) then step i is redundant. To see why, note that i' can implement arbitrary 2-input operators. Whatever function of j and k is computed by i, adding the information j or k will not change the possible functions of j and k that i' can compute. Thus, i' may as well act on inputs j and k directly, and step i becomes unnecessary. This is demonstrated by Figure 2.4.

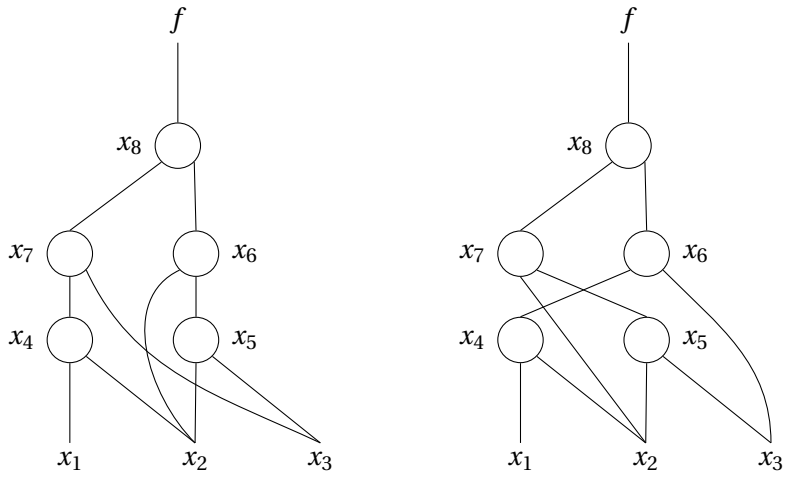


Figure 2.5 – Illustration of symmetries avoided by the (co)-lexicographical symmetry break. Using **(C)**, the topology in (a) would not be valid, whereas the one in (b) would be.

(Co-)Lexicographically ordered steps (C)

Without loss of generality, we may impose a (co-)lexicographical order on the step fanins. In other words, a step like $x_7 = \circ_7(x_3, x_4)$ need never follow a step $x_6 = \circ_6(x_2, x_5)$. To enforce such an order, we can add the clauses $(\bar{s}_{ijk} \vee s_{(i+1)j'k'})$ if $j' < j < k = k'$ or if $k' < k$. Imposing this order significantly reduces the search space by removing all otherwise ordered chains. Fig 2.5 provides an illustration of this symmetry break.

(Co-)Lexicographically ordered operands (O)

Similarly to the previous point, we may enforce an order on step operators as well. We can do this by adding the clauses $((s_{ijk} \wedge s_{(i+1)jk}) \rightarrow f_i \prec f_{(i+1)})$. In this case, we are free to choose a lexicographic or co-lexicographic order, depending on the relation $\prec$.

Ordered symmetric variables (S)

If two function inputs p and q are symmetric ($p < q$), we may ensure that input p is used before q. To do so, we can add the clauses

$$\left(\bar{s}_{ijq} \vee \bigvee_{n<i'<i} \bigvee_{1\leq j'<k'<i'} [\, j' = p \text{ or } k' = p]\, s_{i'j'k'}\right)$$

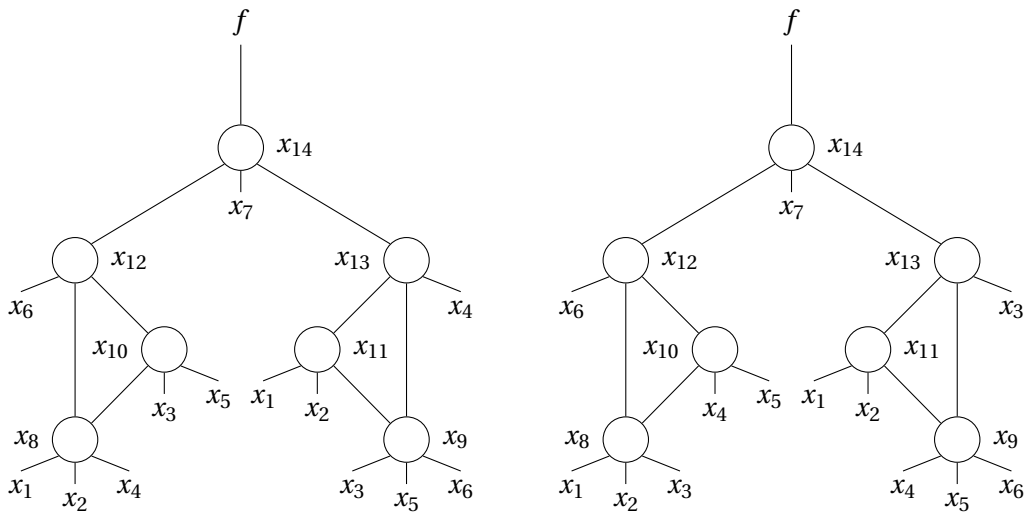


Figure 2.6 – Illustration of symmetries avoided by symmetry break **(S)**. In the topology on the left, we have switched variables x_3 and x_4. Although both topologies are nearly identical minimum-size implementations of f, (a) is invalid under **(S)**, since x_3 must be used before x_4.

whenever $j \neq p$. Figure 2.6 shows an example of this symmetry break, using the majority-7 function $f = \langle x_1, x_2, \ldots, x_7 \rangle$ as an example. It shows a minimum-size topology for this function, using 3-input majority operators.

2.4 Quantitative Comparisons of CNF Encodings

Now that the various encodings and symmetry breaks are defined, we are in a position to perform the experiments in which we compare them. To help us choose which combinations of encodings and symmetry breaks are most useful in practice, we would like to be able to answer the following questions:

1. Which encoding has the smallest runtime on representative benchmarks?
2. What is the impact of various symmetry breaks?
3. Does the answer to (1) change when we increase operator size?

The answer to question (3) tells us if some encodings are better suited for different step operator sizes. This is related to domain suitability, as different domains may require different operator sizes. For example, when synthesizing or mapping into arbitrary-input MIGs we may

wish to use a synthesis engine that is well suited for the synthesis of large operators, whereas this is not the case for AIG synthesis [6, 131].

To implement our experiments we have developed the *percy* library, which is publicly available at https://github.com/whaaswijk/percy. It is part of the *EPFL Logic Synthesis Libraries* [134]. The *percy* library was designed from the ground up to offer a flexibly synthesis interface which can be used to answer questions such as those defined above. The encodings and algorithms of synthesis engines may be quite dissimilar. Moreover, it is not always obvious which combination will be superior in a specific domain. It is often desirable to experiment with several methodologies and SAT solver backends to find the right fit. The aim of *percy* is to provide a flexible common interface that makes it easy to construct a parameterizable synthesis engine suitable for different domains. For a more detailed description of *percy*, see Appendix A.

Using *percy*, our experiments were set up in a generic way, in which the specifications, encodings, and solvers could be configured dynamically. The pseudocode for the synthesis algorithm used in these experiments can be found in Algorithm 1. The algorithm takes as input a specification, and encoder, and a SAT solver. The specification determines which function to synthesize, the encoder determines the encoding to use, and the SAT solver provides the engine on which synthesis is performed. In these experiments, we use the `bsat` solver. All experiments were performed on a machine with a 2x Intel Xeon E5-2680 v3 processor using a 30MB cache and 256 GB DDR4-2133 RAM.

Experiment 1. In this experiment, we synthesize size-optimum 2-input operator Boolean chains for all 65,536 4-input functions. We do so using all three encodings and all 2^6 possible symmetry breaking settings. In other words, for each encoding, we try all possible combination of symmetry breaks, on all 4-input functions. The results of this experiment are summarized in Table 2.1, where we have selected, for each encoding, the two best and the two worst settings with respect to average synthesis runtime. In the symmetries column, a 1 (0) means that a symmetry break was enabled (disabled).

Table 2.1 shows averages for total synthesis runtime, as well as time spent by the SAT solver on SAT and UNSAT CNF formulas. Note that, in this experiment, it is important to control for the time spent generating the encoded CNF formulas. Some encoders may be faster than others up to some constant factor which depends on implementation details. However, we are interested in the merits of the encodings themselves. In other words, we want to compare the difficulty of solving the different CNF formulas and *not* the time taken by some specific implementation to generate them. In practice, good encoder implementations are fast and time spent encoding is negligible: the asymptotic behavior of the synthesis algorithm is determined heavily by the CNF. Therefore, we consider encoding time as noise and measure only time spent by the SAT solver.

Algorithm 1 Basic exact synthesis in *percy*

Require: Specification `spec`
Require: Encoder `enc`
Require: SATSolver `slv`
Ensure: $F(c) \equiv$ `spec.`f

```
 1: procedure SYNTHESIZE
 2:     c ← empty_chain()
 3:     spec.r = 0
 4:     if is_trivial(spec) then
 5:         return c
 6:     end if
 7:     while true do
 8:         spec.r = spec.r + 1
 9:         F_r ← enc.encode(spec, slv)
10:         is_SAT ← slv.solve(F_r)
11:         if is_SAT then
12:             printf("found %d-step solution", spec.r)
13:             c ←enc.extract_chain(F_r, slv)
14:             return c
15:         else
16:             printf("no %d-step solution exists", spec.r)
17:         end if
18:     end while
19: end procedure
```

Table 2.1 – Impact of symmetry breaking on the space of 4-input functions for 2-input operator chains. Sorted by average synthesis time. All times reported in μs.

Encoding	Symmetries						Synth time	SAT time	UNSAT time
	N	A	R	C	O	S			
SSV	1	1	1	1	0	1	346.35	148.74	197.61
SSV	1	1	1	1	1	1	384.89	173.96	210.93
MSV	1	1	1	1	0	1	454.51	145.00	309.52
MSV	1	1	1	1	1	1	486.77	196.50	290.19
DITT	1	1	1	1	0	1	576.11	207.53	368.57
DITT	1	0	1	1	0	1	584.19	195.86	388.33
DITT	0	0	0	0	1	0	3,062.35	460.96	2,601.38
DITT	0	1	0	0	1	0	3,256.90	423.97	2,832.93
MSV	1	0	0	0	1	0	4,038.83	506.24	3,532.59
MSV	0	0	0	0	1	0	4,191.09	441.24	3,749.67
SSV	0	0	0	0	0	0	4,025.28	693.94	3,385.34
SSV	0	0	0	0	1	0	4,414.15	647.29	3,766.85

First, let us consider the impacts of symmetry breaking. The results show that symmetry breaks have a very significant impact on runtime. For example, the best SSV encoding enables most symmetry breaks and is more than **10x** faster than the worst two, which disable (almost all of) them. We see similar behavior for the MSV and DITT encodings as well. Their best settings are more than **9x** and **5.5x** faster than their worst settings, respectively. Next, let us look at the differences between encodings. The best SSV encoding is **24%** and **40%** faster than the best MSV and DITT encodings, respectively. Thus, we see that the choice of encoding and symmetry breaks has a notable impact on synthesis runtime.

Experiment 2. In the next experiment, we investigate question (3) by measuring runtime while increasing the number of inputs as well as Boolean chain operator size. Therefore, we now to synthesize 5-input functions using Boolean chains with 3-input operator steps. The space of 5-input functions is too large to run this experiment on all of them. Instead, we use NPN canonization and synthesize only the first 222 5-input NPN classes. Table 2.2 contains the summary of results.

We again find the SSV encoding the be fastest, although the gap with the other encodings appears to be closing. It has reduced to **18%** and **31%** with respect to the best DITT and MSV encodings, respectively. We see again that symmetry breaking settings are quite significant, with difference of **39%**, **1.5x**, and **3.3x** between the best and worst SSV, MSV, and DITT encodings, respectively.

Experiment 3. To further investigate the impact of different encodings on input and operator scaling, we test on a set of 500 non-DSD decomposable 6-input functions. These functions were harvested from the MCNC/ISCAS/ITC benchmark suites and should therefore be repre-

Table 2.2 – Impact of encoding and symmetry breaking for 5-input functions with 3-input operator chains.

Encoding	Symmetries						Synth time	SAT time	UNSAT time
	N	A	R	C	O	S			
SSV	1	0	1	0	0	1	2,390.83	1,598.42	792.40
SSV	0	1	1	0	0	1	2,513.68	1,705.35	808.33
DITT	0	0	0	0	0	1	2,901.33	1,819.91	1,081.42
DITT	0	1	0	0	1	0	2,908.45	1,864.55	1,043.89
MSV	1	1	1	0	0	1	3,459.78	2,237.28	1,222.50
MSV	0	1	1	0	0	1	3,590.21	2,289.83	1,300.38
SSV	0	0	1	1	1	0	3,940.55	2,968.78	971.77
SSV	0	1	0	1	1	0	3,949.33	2,904.07	1,045.26
MSV	0	0	0	0	0	0	5,293.76	3,722.19	1,571.58
MSV	1	1	0	1	1	0	5,312.44	3,830.85	1,481.60
DITT	1	1	0	1	1	0	9,573.93	7,120.95	2,452.97
DITT	0	0	0	1	1	0	9,619.08	7,218.89	2,400.20

sentative of functions which appear in concrete circuits. We now perform synthesis for chains with 4-input operators. Such large operators are used in (re-)synthesis and mapping of k-LUTs. Results are reported in Table 2.3.

We see that the MSV and DITT encodings are now starting to outperform the SSV one. They are **15%** and **11%** faster, respectively. This is likely caused by the selection variable scaling described above. As the chain operator size increases, so do the number of possible fanin combinations. Since the number of selection variables in the MSV and DITT encodings scales linearly, we expect these encodings to be more efficient than the SSV one, which scales quadratically. Again, there are significant differences between the best and worst symmetry breaking settings of encodings. The runtime difference is **1.74x**, **74x**, and **29%** for the MSV, DITT, and SSV encodings respectively.

These experiments clearly show that, given an exact synthesis problem, the choice of encoding and symmetry breaks has a great impact on the expected runtime. The best choice depends heavily on both the function domain and operator size. Runtime differences between different encodings can be significant (up to **31%**), but the largest impact is due to symmetry breaking within encodings (up to **74x**). Interestingly, enabling more symmetry breaks does not guarantee improved runtimes. We conjecture that this is due to the fact that many symmetry breaks were developed in the context of 2-input operator synthesis. Therefore, they may not scale to the general case. Some, such as ***(S)***, seem to apply more universally and reduce runtime in all or most cases. Note that this kind of symmetry is independent of fanin size.

Table 2.3 – Impact of encoding and symmetry breaking for 6-input functions with 4-input operator chains.

Encoding	Symmetries						Synth time	SAT time	UNSAT time
	N	A	R	C	O	S			
MSV	1	1	1	0	0	1	72,074.97	46,137.25	25,937.72
MSV	1	1	1	0	0	0	73,424.64	47,332.30	26,092.34
DITT	1	0	0	0	0	1	74,877.43	45,847.62	29,029.81
DITT	0	0	0	0	0	1	75,604.72	44,941.67	30,663.04
SSV	0	1	1	0	0	1	84,351.44	52,447.10	31,904.33
SSV	0	1	1	0	0	0	84,401.24	52,449.83	31,951.41
SSV	0	0	0	1	1	0	118,156.39	88,156.71	29,999.68
SSV	1	1	1	1	1	0	118,670.60	91,156.27	27,514.33
MSV	1	1	0	1	1	1	122,134.72	96,746.38	25,388.35
MSV	1	1	0	1	1	0	125,737.32	100,220.61	25,516.71
DITT	1	0	0	1	1	0	5,482,999.48	4,554,896.87	928,102.61
DITT	0	0	0	1	1	0	5,545,923.66	4,617,253.56	928,670.09

2.5 CEGAR

The concept of *counterexample-guided abstraction refinement* (CEGAR) was first introduced by Clarke et al. in [32]. Originally designed to mitigate the state explosion problem in verification algorithms, CEGAR is a generic solving technique. It can be applied to SAT solvers as well other types of solvers that operate on higher levels of abstraction, such as those used in *bounded model checking* (BMC) [19, p.474]. Broadly speaking, abstraction refinement works as follows.[2] Initially, given a problem instance, we remove some of the constraints that would normally be used to guarantee correctness. In doing so, we therefore under-constrain our solver. Thus, if we cannot find a solution for the under-constrained problem, we are sure that no model exist for the actual problem, since more constraints can only result in fewer solutions. We then invoke the solver on the under-constrained problem. After some time, the solver will respond with either true or false. If the response is false, then we *know* that no solution for our original problem exists. On the other hand, if the response is true, then a solution *may* exist. We verify if the given solution is valid for our actual constraints. If it is, we have found a valid solution and we are done. If it is not, we find a counterexample where the solution fails. Using the counterexample, we add additional constraints such that the solver will not again find a solution that fails in the same way. We repeat these steps until either we obtain a valid solution (i.e no more counterexamples exist) or we find that no such solution exists.

CEGAR can be used to speed up constraint solver problems. By not providing all constraints from the start, we can construct smaller initial problems. In the context of SAT, this corresponds to formulae with fewer variables and clauses. If we find that the problem is impossible at an

[2] In fact, what we describe here is the conceptual dual of the typical CEGAR formulation. However, it is more clear and appropriate for our purposes here. We refer the interested reader to [32] and [19] for more details.

early stage, we can terminate early. Moreover, we can terminate if we find a valid solution for which no counterexamples exist. Hence, we may terminate early in both cases without having to construct the full set of constraints. This approach does rely on the ability to efficiently find and add counterexamples to the solver.

There are various ways that one may apply CEGAR to SAT-based exact synthesis. We describe here a technique that is commonly used in practice. Recall the synthesis problem expressed by the tuple $(f, \mathscr{B}, r)$. When synthesizing a Boolean chain we can apply CEGAR to the minterms of f. To do so, we generate a partial encoding $\mathscr{F}_r'$. Exactly what variables and clauses the partial encoding contains depends on the specific encoding which is used. The standard encoding $\mathscr{F}_r$ contains all constrains to ensure that synthesized chains c agree with f on all minterms. In other words, it guarantees $F(c) = f$. The encoding $\mathscr{F}_r'$, on the other hand, only provides this guarantee for a subset of minterms S. More precisely, it guarantees $t \in S \rightarrow c(t) = f(t)$. There is no guarantee that c will compute the correct simulation value for any minterms outside of S. However, in practice it turn out that in many cases we do not need to provide constraints for all minterms in order to find a chain that satisfies the full specification for f. Therefore, CEGAR-based exact synthesis can take advantage of smaller under-constrained CNF formulae which use fewer variables and clauses, and still find valid solutions. Algorithm 2 shows the pseudocode for a CEGAR-based extension of Algorithm 1.

Algorithm 2 creates an initial CEGAR encoding on line 9. This partial encoding does not provide any guarantees for correctness. The algorithm then proceeds to add minterm constraints on line 11. The initial minterm for which constraints are added is the 0 vector. Depending on the specific problem domain, one may wish to choose a different initial minterm. However, we follow this convention here, since it does not matter for correctness and it simplifies the example. After adding the minterm constraints, the SAT solver is invoked on the partial encoding to find a solution. If the problem is UNSAT, we know that no solution with r gates exists. Otherwise, if the problem is SAT, we have a solution for the partial encoding. This solution corresponds to a chain that will partially agree with the specification. The function computed by this chain is found by simulating it, resulting in a truth table representation. By taking the bitwise difference between the specified function and the truth table of the chain, we can efficiently find minterms for which the chain computes the wrong value. If such minterms exist, they are counterexamples for the correctness of the chain. We store the vector index for the first bit of difference. If no such index exist there are no counterexamples. In that case we store the `NULL` index -1. We iterate the above process, gradually enlarging set S, until no more counterexamples are found; a situation which is represented by storing the `NULL` index. In doing so, we are guaranteed to eventually find a solution that is correct on all minterms.

To show the runtime impact of CEGAR-based synthesis, we present another experiment. In

Algorithm 2 Synthesis algorithm based on a CEGAR loop

Require: Specification `spec`
Require: Encoder `enc`
Require: SATSolver `slv`
Ensure: $F(c) \equiv$ `spec`$.f$

```
1:  procedure CEGAR_SYNTHESIZE
2:      c ← empty_chain()
3:      spec.r = 0
4:      if is_trivial(spec) then
5:          return c
6:      end if
7:      while true do
8:          spec.r = spec.r + 1
9:          F'_r ← enc.init_cegar_encoding(spec)
10:         for t = 0; t != -1; t = t+1 do
11:             F'_r ← enc.add_minterm_constraints(spec, slv, t)
12:             is_SAT ← slv.solve(F'_r)
13:             if is_SAT then
14:                 truth_table = enc.simulate(F'_r, spec, slv)
15:                 t = bit_difference(spec, truth_table)
16:             else
17:                 printf("no %d-step solution exists", spec.r)
18:                 break
19:             end if
20:         end for
21:         if t == -1 then
22:             printf("found %d-step solution", spec.r)
23:             c ←enc.extract_chain(F'_r, slv)
24:             return c
25:         end if
26:     end while
27: end procedure
```

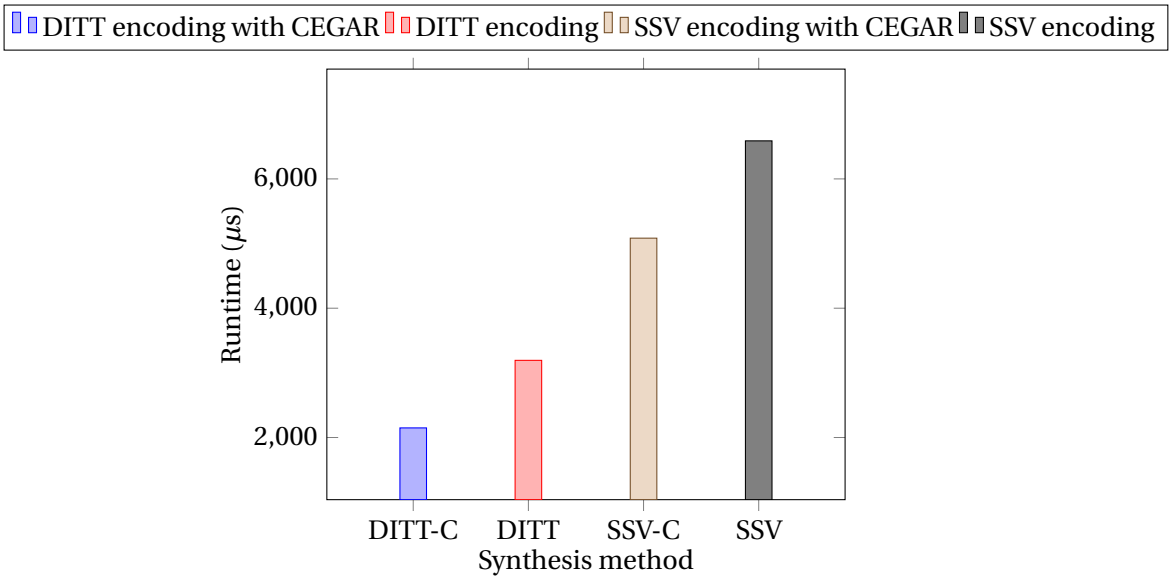


Figure 2.7 – An illustration of the impacts of CEGAR on synthesis runtime.

this case, we decompose a 5-input majority function into an optimum-size Boolean chain of 3-input majorities. Such a decomposition can be done by slightly altering the encodings described in this chapter. Essentially, one just needs to allow for 3-input operators, and restrict the operators to represent only 3-input majority functions. We have created such alternative encodings for both the SSV and DITT encodings. The results are shown in Figure 2.7. It shows, for both encodings, a comparison of the runtimes in μs obtained with Algorithms 1 and 2, respectively. The first thing we can see is that the DITT encoding seems to be more suited for the synthesis of majority-3 chains than the SSV encoding, as the fastest DITT encoding is roughly **2x** faster than the fastest SSV encoding. Moreover, for both encodings the use of CEGAR has a significant impact. Using CEGAR reduces runtime by **23%** and **33%** for the SSV and DITT encodings, respectively. Thus, this experiment again confirms the importance of testing different encodings for each new synthesis domain, as the right choice of encoding makes a large difference in runtime. Moreover, we see that adding a CEGAR loop can be a useful addition to the core synthesis algorithm. Therefore, in most practical scenarios, it is the preferred default method to use over a monolithic one-shot encoding.

2.6 Synthesis With Don't Cares

The encodings presented in this chapter can be adapted to take don't care conditions into account. This is useful in applications such as logic rewriting, where we know (e.g. due to structural constraints imposed by the network) that certain input patterns never occur or that

outputs are not observable under some conditions. Let us start by formally defining the exact synthesis problem with don't cares, which can be viewed as a generalization of the definition in Section 2.1.2. We are given an incompletely specified Boolean function $f^* : \{0,1\}^n \to \{0,1,*\}^m$. Let DC be the don't care set for f. Given $(f, \mathscr{B}, r, DC)$, we can now define the question $\mathscr{Q}_r^*$:

> "Does there exist a Boolean chain c such that $\sigma(c) = r$, $F(c) = g$, $\omega(c) \subseteq \mathscr{B}$, and $f \oplus g \subseteq DC$?"

In other words, we want to synthesize a chain with r gates such that all differences between the chain function and the specified function fall within the don't care set.

It is straightforward to extend our other synthesis algorithms to use don't cares. We now only have to add truth table constraints for those minterms that fall within the care set. For all other minterms, by definition we do not care what the output value of the Boolean chain is. The extended algorithm can be found in Algorithm 3.

Algorithm 3 receives a don't care mask as an additional input. This mask is simply a truth table whose bits are set to 1 on minterms in the don't care set. The algorithm proceeds by entering a CEGAR loop, similar to the one in Algorithm 2. However, to compute bit indices for counterexamples, it now computes the bitwise AND of the difference truth table and the *negation* of the don't care mask. As a result of this computation, bits in the difference truth table cannot be 1 if they fall in the don't care set. Hence, this is an efficient way of ensuring that differences between the chain function and the specified function are ignored if they fall in DC. The algorithm algorithm could be adapted to work with a care set by simply removing the negation in line 16.

2.7 Computational Complexity

To finish this chapter, we analyze the computational complexity of SAT-based exact synthesis. However, before we delve into the specifics, it will be instructive to take a look at the historical context of the exact synthesis problem, as it has been studied extensively by theoretical computer scientists.

Recall that, given the tuple $(f, \mathscr{B}, r)$, we have defined the exact synthesis problem as the following question $\mathscr{Q}_r$:

> "Does there exist a Boolean chain c such that $\sigma(c) = r$, $F(c) = f$, and $\omega(c) \subseteq \mathscr{B}$?"

In turns out that $\mathscr{Q}_r$ is in fact a special case of what, in theoretical computer science, is known as the *minimum circuit size problem* (MCSP) [102, 4]. The MCSP is: *given the truth table of*

Algorithm 3 Exact synthesis with don't care support

Require: Specification `spec`
Require: Encoder `enc`
Require: SATSolver `slv`
Require: TruthTable `dc_mask`
Ensure: $F(c) \oplus$ `spec` $\subseteq$ `dc_mask`

```
1:  procedure DC_SYNTHESIZE
2:      c ← empty_chain()
3:      spec.r = 0
4:      if is_trivial(spec) then
5:          return c
6:      end if
7:      while true do
8:          spec.r = spec.r + 1
9:          F'_r ← enc.init_cegar_encoding(spec)
10:         for t = 0; t != -1; t = t+1 do
11:             F'_r ← enc.add_minterm_constraints(spec, slv, t)
12:             is_SAT ← slv.solve(F'_r)
13:             if is_SAT then
14:                 truth_table = enc.simulate(F'_r, spec, slv)
15:                 xor_tt = truth_table ⊕ spec
16:                 xor_tt = xor_tt ∧ ¬dc_mask
17:                 t = first_one_bit(xor_tt)
18:             else
19:                 printf("no %d-step solution exists", spec.r)
20:                 break
21:             end if
22:         end for
23:         if t == -1 then
24:             printf("found %d-step solution", spec.r)
25:             c ←enc.extract_chain(F'_r, slv)
26:             return c
27:         end if
28:     end while
29: end procedure
```

a Boolean function f and a size parameter r, is the combinational complexity of f at most r? Typically, the MCSP is defined for circuits over AND, OR, and NOT gates of fanin at most 2, which roughly correspond to Boolean chains with fanin-2 steps. Formally, in the MCSP problem we are given a tuple (T, r), where T is a string of $N = 2^n$ bits (a truth table), and r is a positive integer.

The MCSP is considered a fundamental problem in computer science and research on it dates back at least to the 1950s in the USSR [146]. The remaining interest in the problem stems partly from the fact that it is such a natural question, and partly from the connections that it has to other problems in complexity theory. As we analyze the runtime of our algorithm below, we will find it to be quite high. However, as we will see, this is not surprising given the complexity results that have been found for the MCSP.

Let us begin our analysis by computing some asymptotic lower and upper bounds on the combinational complexity of Boolean functions. It is widely known that we can implement any arbitrary function f on n variables with at $r = O(2^n)$ gates. This follows directly from Boole's expansion, which tells us that

$$f(x_1, x_2, \ldots, x_n) = (x_1 \wedge f(1, x_2, \ldots, x_n)) \vee (\bar{x}_1 \wedge f(0, x_2, \ldots, x_n)).$$

Hence, anytime we expand a function, we need one OR gate, two AND gates, and one NOT gate. Thus, an upper bound G on the number of gates for f is given by the following recurrence relation:

$$G(n) = 2G(n-1) + 4$$
$$G(2) = 1$$

Solving this recurrence yields $G(n) = \frac{5}{4}2^n - 4 = O(2^n)$.[3] A tighter upper found, due to Lupanov, is $(1+o(1))\frac{2^n}{n}$. Hence, we have an exponential upper bound for the size of any circuit. Moreover, due to Shannon's counting argument, we also know that this upper bound is tight. Indeed, we know that almost every Boolean function requires at least $\frac{2^n}{n}$ gates [124].

Now that we have established the proper upper and lower bounds, we can easily check that the MCSP is in NP. To see why, note that we can nondeterministically guess a circuit of size r in time $O(r \log r)$ [102]. Moreover, due to the upper bound we have computed above, we know that if $r > c2^n$ (for some small constant c), then the MCSP answer is trivially yes. Let us therefore assume that $r \leq 2^n = N$. Then, we have $O(r \log r) \leq O(\text{poly}(N))$. This means that we can guess an appropriate circuit in time polynomial in N. Next, we can verify that the circuit is

[3] Note that G(1) is either zero or one, depending on how we count inverters.

correct by simply simulating it on all minterms, and verifying its outputs with the specified truth table T. We can clearly do this in poly(N, r) time. Thus, there exist a non-deterministic Turing machine that can decide the MCSP problem in polynomial time, and hence MCSP $\in$ NP.

Interestingly, it is currently unknown if, in addition to being in NP, the MCSP is also NP-hard. Recall that a problem P is NP-hard if there exists a polynomial reduction from any problem $P' \in$ NP to P. Work by Kabanets and Cai has shown evidence that finding "natural" polytime reductions to MCSP is likely to be difficult [67]. Later work by Murray and Williams provides similarly evidence [102]. The evidence provided in these works roughly takes the following form: if some type of reduction would exist, then this would imply either a separation or a collapse of some fundamental computational complexity classes. Since proving such separations, or collapses, is suspected to be very hard to do, this is taken as evidence that the NP-hardness of the MCSP will be hard to prove as well. Conversely, it is unknown if MCSP $\in$ P, although there is plenty of strong evidence to suggest that it is not [67]. Generally speaking, most experts agree that the computational complexity of the MCSP likely to be superpolynomial.

Now that we have established the apparent hardness of the MCSP problem, let us go back to considering the computational complexity of answering $\mathscr{Q}_r$ using SAT. We will analyze it in a way that is similar to the analyses that have been made of the MCSP. We define the input size of the problem to be $N = 2^n$, where n is the number of function inputs. Note that it is quite natural to take the truth table as our input size rather than the number of function inputs n: any algorithm that synthesizes a circuit based on a truth table specification, certainly requires at least the time needed to read the specification.

Recall that, given a CNF formula ϕ on k variables, the worst-case runtime for SAT is $O(|\phi|2^k)$ [19, Chapter 1], where $|\phi|$ is the length of the formula. [4] As we have seen in Section 2.2, given $\mathscr{Q}_r$, all encodings we have discussed generate CNF formulas with $O(\text{poly}(N, r))$ variables. Therefore, the expected runtime for our SAT-based algorithm is

$$O(|\mathscr{F}_r|2^{\text{poly}(N,r)}) = O(|\mathscr{F}_r|2^{\text{poly}(2^n,r)})$$

Hence, noting that $|\mathscr{F}_r| = O(\text{poly}(N, r))$, the worst-case runtime of our algorithm is in fact doubly exponential in n.

It is also interesting to consider the asymptotic average case runtime of our algorithm. We know that, for large n, most functions circuits require circuits with $r \geq \frac{2^n}{n}$. Hence, the expected

[4] To the best of our knowledge at the time of writing. Any polynomial time algorithm would of course imply P = NP.

computational complexity of finding a circuit for an arbitrary function is

$$O(|\mathscr{F}_r| \cdot 2^{\text{poly}(2^n, \frac{2^n}{n})})$$

In other words, in the average case, we expect synthesis to require an exponential number of gate variables (e.g. selection variables, step operator variables). This, in turn, implies that the asymptotic expected runtime of our algorithm is doubly exponential in n. Furthermore, depending on our search strategy, our algorithm may also be required to construct an exponential number of CNF formulas $\mathscr{F}_r$. That is, if we start with $r = 0$ and increment r until we find a value that works, as described in Section 2.1.2, we would not expect to find such a value until $r \sim \frac{2^n}{n}$. Hence, for large n we would likely wish to change our search strategy. However, this largely a moot point, as we do not expect any SAT algorithm to be applicable to most problem instances when n becomes very large.

2.8 Summary

In this chapter, we have presented the foundations of SAT-based exact synthesis, focusing on methods for synthesizing size-optimum Boolean chains. We discussed standard terminology and notation, as well as pointed to the existing literature. We also provide detailed analyses of three commonly used CNF encodings and symmetry breaks that can be used to reduce the SAT search space. With this information in hand, we have presented a number of experiments that quantitatively compare the encodings as well as the interplay between encodings and symmetry breaking. We found that there are significant differences in runtime between encodings, and that the notion of "best encoding" is domain dependent. Moreover, we found that the use of proper symmetry breaking constraints also has a significant impact on runtime, and that it is not the case that adding more symmetry breaks necessarily reduces runtime. Finally, we discussed three different synthesis algorithms, based on different solving techniques: (i) the basic algorithm based on monolithic CNF formulae, (ii) synthesis based on CEGAR, and (iii) an extension of CEGAR-based synthesis with support for don't cares. In conclusion, this chapter contains the information necessary for the construction of efficient and general purpose SAT-based exact synthesis algorithms. In the next chapter, we show how the addition of DAG topology families can be used to unlock further efficiency gains and parallel synthesis algorithms.

3 DAG Topology Families

SAT-based exact synthesis has always contended with unpredictable, and potentially slow runtimes. This is perhaps unsurprising if we consider that, in finding optimum Boolean chains, the SAT solver has to simultaneously perform at least two distinct tasks:

1. finding valid DAG structures for the Boolean chain
2. assigning Boolean operators to the vertices in these DAGs, such that the resulting chain realizes the specified Boolean function

Topology-based synthesis is a proposal to mitigate the difficulty of step (1), or to avoid it altogether. In topology-based synthesis, we augment SAT-based exact synthesis with DAG topology information. Thus, we shrink the SAT solver's search space by providing additional, domain-specific, knowledge. The goal of this new approach is to reduce synthesis runtime, and to take a step towards unlocking the potential of exact synthesis.

3.1 Introduction

When a solver is supplied with the appropriate DAG topology information, the exact synthesis problem is greatly simplified. Suppose we are given a DAG $G = (V, E)$, and a Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$. We may be able to transform the DAG into a Boolean chain for f by assigning the appropriate operators $\phi_i \in \mathscr{B}$ to every vertex $v_i \in V$. We call such a transformation a *labeling* of the graph. Finding such a labeling may not be possible, but if it exists, a SAT solver can find it efficiently. For example, consider the single-output 6-input function with truth table 0x9ef7a8d9c7193a0f.[1] The smallest known implementation of this function uses 19 2-input

[1] For conciseness, we represent the binary truth table as a hexadecimal string where the right-most characters represent the least significant bits.

gates. We can extract the underlying DAG structure from this 19 gate solution. When it is given, is given, a SAT solver can find a labeling in 0.12s on a laptop computer. Without this topology, finding a solution is intractable. It is currently unknown if there is a chain for this function that uses fewer than 19 steps. The above solution was obtained using a combination of Boolean decomposition and circuit enumeration, rather than exact synthesis techniques.

The efficiency of labeling may inspire one to think of a (naive) synthesis algorithm which, given f, simply enumerates DAG structures until it finds one that can be labeled. Such an algorithm reduces to efficiently finding a DAG with the proper structure for f. However, in general, given f we do not know a priori which DAG structures have a labeling. Given an n-input function, finding a suitable DAG requires us to search a very large space of DAG structures. Unfortunately, the enumeration of potential DAGs in this space generally outweighs the potential efficiency of graph labeling. To see why, we can refer to the first column of Table 3.1, which contains the numbers of DAGs up to 12 vertices.

Alternatively, we can specify a set of clauses which constrain the SAT solver's search to a particular family of DAG topologies. We then use the SAT solver's efficient search heuristics to find only those topologies within that family. This approach avoids explicit enumeration of DAGs and provides a middle ground between the unstructured exact synthesis formulation of Section 2.1 on the one hand, and the fully structured labeling of graphs on the other hand. In Sections 3.2 and 3.3 we introduce two different types of topology families. Both explore this middle ground in different ways and can be used to achieve significant runtime improvements over conventional unstructured encodings.

3.2 Fences

Given two integers k and l ($1 \leq l \leq k$), a *Boolean fence* is a partition of k nodes over l levels, where every level contains at least one node. We can denote a Boolean fence by a sequence $F = (\lambda_1, \ldots, \lambda_l)$, where every λ_i corresponds to the number of nodes on level i, with the additional constraints

$$\sum_{i=1}^{l} \lambda_i = k$$

and

$$\lambda_i \geq 1 \text{ for all } 1 \leq i \leq l.$$

A Boolean fence (k, l) is not unique: there may be multiple ways of distributing k nodes over l levels. We call the set of all such partitions a Boolean fence *family* and write $\mathscr{F}(k, l)$. We use

$\mathscr{F}_k$ to denote the set of all fence families of k nodes:

$$\mathscr{F}_k = \{\mathscr{F}(k,l) \mid 1 \leq l \leq k\}$$

To be concise, we also refer to Boolean fences and fence families as fences and families, respectively.

Boolean fences can be visualized as graph topologies without edges. Figure 3.1 shows $\mathscr{F}_k$ for $1 \leq k \leq 5$. In each drawing we show the node distribution of a fence across different levels. Adjacent fences are drawn in different colors to make them easier to distinguish.

Every DAG of n nodes corresponds to a unique fence $F \in \mathscr{F}_n$. To see why, note that we can assign levels to nodes in a DAG based on their partial order. Such an assignment allows us to find the level distribution corresponding to the fence F.

A fence induces a set of DAG topologies, in which each topology corresponds to the same distribution of nodes over levels, but with different arcs between nodes. In other words, fences represent families of graph topologies. Consequently, a fence induces a set of Boolean chains with those topologies.

3.3 Partial DAGs

Fences are one type of topology family which can be used to add some additional structure to SAT-based exact synthesis. However, they still leave a fair bit of structure unspecified. For instance, they do not specify any connections between steps. Moreover, they are even agnostic with respect to the number of possible fanins of each node. In some scenarios this flexibility may be desirable. However, in others we might benefit from additional structure. For instance, we may know that we want to synthesize Boolean chain with 2-input operators up to some number r steps. Preferably, our synthesis method would be able to take advantage of this information.

A *partial DAG* is a topological structure which may be viewed as a partial specification of the underlying DAG structure for a Boolean chain. It specifies two things: (i) the number of fanins for each step, and (ii) the connections between internal nodes. All connections to primary inputs are left unspecified. Note that one can recover a level distribution from the internal connections of a partial DAG. Hence, since they also specify internal fanin connections, partial DAGs contain strictly more structural information than fences.

More formally, a partial DAG of n nodes can viewed as a sequence of k-steps:

$$(x_{11}, x_{12}, \ldots, x_{1k}), \ldots, (x_{n1}, x_{n2}, \ldots, x_{nk})$$

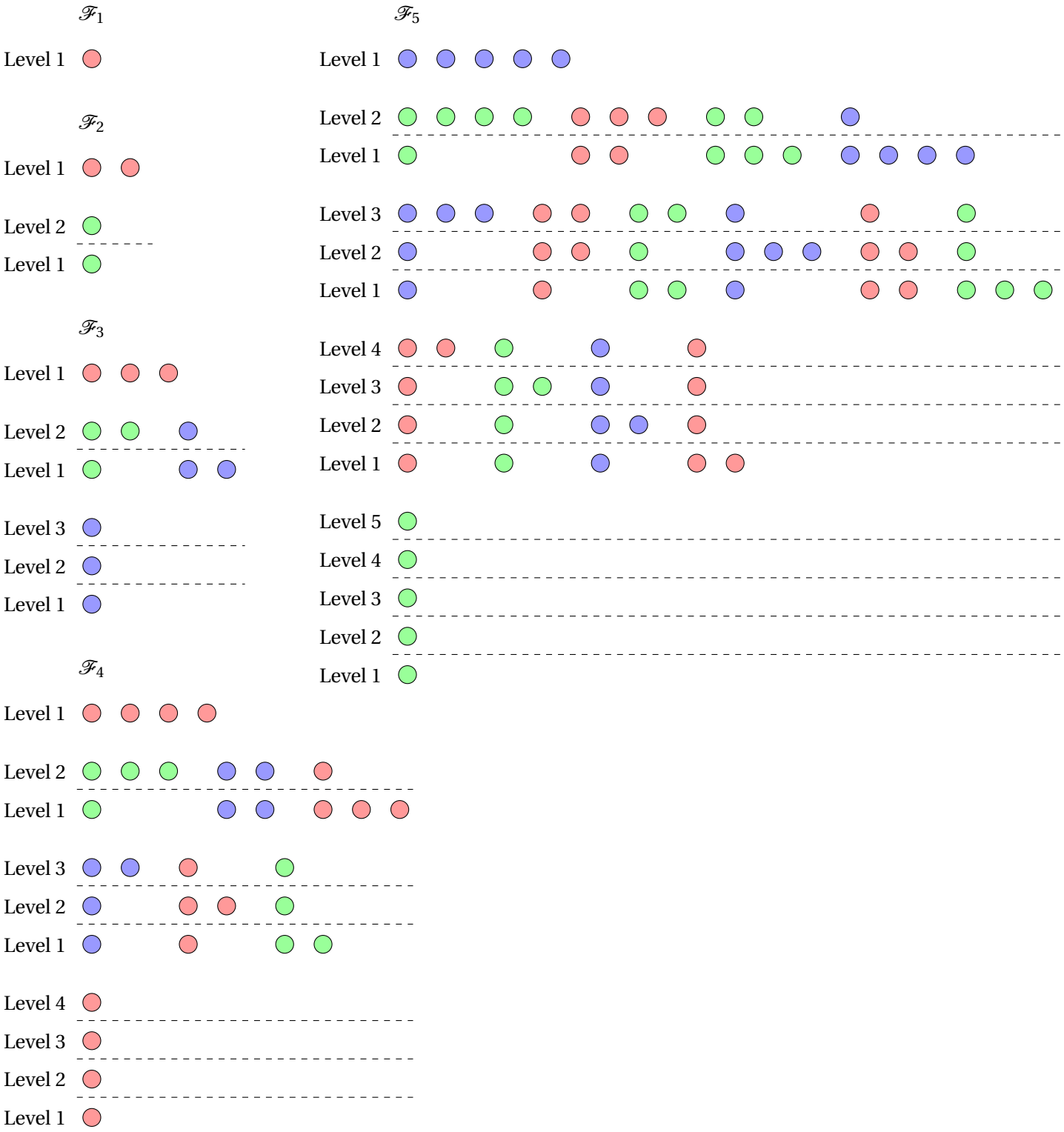


Figure 3.1 – Illustrations of the first five fence families.

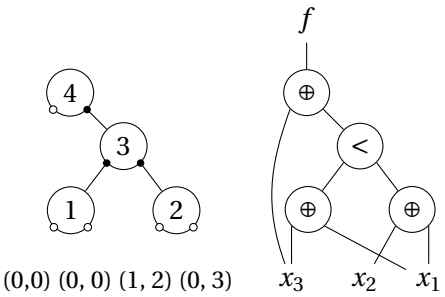


Figure 3.2 – On the left an example of partial DAG specified by the sequence below. Unspecified fanins are signified by empty circles. On the right a fully specified chain found by the SAT solver for the function $f = \langle x_1 x_2 x_3 \rangle$.

If $x_{ij} = 0$ ($j < i$), then the j-th fanin of step i points to some unspecified primary input. Otherwise, if $x_{ij} = m$ ($m < i$), then the j-th fanin of step i points to the m-th step in the chain. Figure 3.2 shows an example of a partial DAG and the corresponding sequence of steps. Note that, like fences, partial DAGs are agnostic with respect to the number of primary inputs they should be synthesized with.

We can efficiently generate (and filter) partial DAGs through a recursive backtrack search algorithm, similar to a fence-generating algorithm. Additionally, we can perform SAT-based exact synthesis using partial DAGs in a similar way to fence-based synthesis, reducing the size of CNF formulas through the structural information encoded in the DAGs.

3.4 Counting Dags, Fences, and Partial DAGs

Let us consider the following question: how many fences are there in family $\mathscr{F}(k,l)$? Note that, in this family, l nodes are fixed, since we need to have at least one node on l levels. The remaining $k-l$ nodes may be arbitrarily distributed across the l levels. In other words, our question reduces to: how many ways are there to distribute $k-l$ indistinguishable nodes across l bins? The answer is equal to the number of nonnegative integer-valued solutions to the equation

$$x_1 + x_2 + \cdots + x_l = k - l$$

and hence

$$|\mathscr{F}(k,l)| = \binom{k-1}{l-1}. \tag{3.1}$$

We can now use Formula 3.1 to count the total number of fences of k nodes, $|\mathscr{F}_k|$ as follows:

$$|\mathscr{F}_k| = \sum_{i=1}^{k} \binom{k-1}{i-1} = 2^{k-1}$$

The reader may verify that these formulas correctly compute the numbers of fences in Figure 3.1. This formula for the number of fences confirms our intuition. Although the number of fences grows exponentially, it is still many orders of magnitude less than the number of DAGs (see Table 3.1). Moreover, there are some other techniques we can use to reduce the number of fences that are "relevant" to a given synthesis problem. For instance, if we want to synthesize a single-output function, we may disregard all fences that have more than one node on the top level. Similarly, if we know that the operators in a chain we want to synthesize have fanin 2, we may disregard fences that have more than two nodes directly below the top level. Through this process, which we call *filtering* we can further reduce the number of fences that we need to consider. In Table 3.1 we show the number of fences needed for the common problems of synthesizing single-output functions for chains with 2- and 3-input operators. We write Fences x/y to signify the number of filtered fences relevant to x-output functions and chains with y-input operators.

Counting the number of partial DAGs is slightly more involved as it depends on the fanin size k. We show here a derivation for the number of partial DAGs with fanin size 2. Obviously, there is only 1 partial DAG with 1 node. It consists of the single step sequence $(0,0)$ since the node may only point to primary inputs. In a partial DAG with 2 nodes, the second node may either point to two primary inputs, or select a primary input and the first node. Similarly, a third node could either point to two primary inputs, or select a primary input and the first node, a primary input and the second node, or select both preceding steps. From the pattern that arises we can see that generally the n-th node has $1 + \binom{n}{2}$ possible fanin options: either it has two primary input fanins, or it may select 2 distinct fanins from the n-element set of previous steps and primary inputs. Therefore, the possible number of n step partial DAGs F_n is given by the formula

$$F_n = \prod_{i=1}^{n} (1 + \binom{i}{2})$$

where we follow the convention that $\binom{1}{2} = 0$.

Table 3.1 shows the number of partial DAGs up to 12 nodes (Unfiltered PD/2). We write PD/k for the number of partial DAGs with k-fanin steps. While the number of partial DAGs is orders of magnitude smaller than the total number of DAGS, it is still quite large. Fortunately, we can perform a number of filtering steps. For example, we may use some of the symmetry breaks described in Section 2.3 to reduce the number of DAG topologies. Furthermore, for any set of

Table 3.1 – Comparing the numbers of DAGs, partial DAGs, and fences for increasing numbers of vertices.

Nr. of vertices	DAGs	Unfiltered PDs/2	Filtered PDs/3	Filtered PDs/2	Fences	Fences 1/3	Fence 1/2
1	1	1	1	1	1	1	1
2	3	2	1	1	2	1	1
3	25	8	3	3	4	2	2
4	543	56	15	9	8	4	3
5	29,281	616	45	41	16	7	6
6	3,781,503	9,856	383	235	32	14	12
7	1,138,779,265	216,832	3,512	1,660	64	28	23
8	783,702,329,343	6,288,128	33,696	13,961	128	56	45
9	1,213,442,454,842,881	232,660,736	344,691	136,875	256	112	90
10	4,175,098,976,430,598,143	10,702,393,856	3,701,536	1,536,631	512	224	180
11	31,603,459,396,418,917,607,425	599,334,055,936	41,204,800	19,484,561	1,024	448	360
12	521,939,651,343,829,405,020,504,063	40,155,381,747,712	472,131,247	275,949,886	2,048	895	719

isomorphic partial DAG topologies, we may select one representative and remove the others. In our experiments, we use the *Nauty* package to efficiently find isomorphic partial DAGs [86]. Here, we are helped by the fact that all nodes in an n node partial DAG with k-steps have bounded degree. We can find isomorphisms between DAGs of bounded degree in polynomial time [83]. Table 3.1 also shows the number of filtered partial DAGs for 2-steps and 3-steps. These numbers are again orders of magnitude smaller than the total number of partial DAGs (of 2-steps, and 3-steps, respectively). Indeed, the numbers are small enough that they may be kept in memory, stored on disk, or in a database. When compressed all the partial DAGs up to 12 nodes for 2-steps take up less than 1GB of space.

3.5 Generating Fences

As we have seen, fences are simple combinatorial structures that are easy to count. It is therefore perhaps unsurprising that generating them is also simple and can be done efficiently. In this section, we describe algorithms based on integer partitioning and recursive backtracking, both of which can be used to efficiently generate streams of fence structures. Both of these methods have been implemented in *percy*.[2]

3.5.1 Integer Partitioning Method

Suppose we want to generate $\mathscr{F}_k$. In order to do this, we first observe that the number of fence families in $\mathscr{F}_k$ closely corresponds to different integer partitionings of k. Recall that, given an integer k, an integer partition of k is a way of writing k as the sum of positive integers $k_1 + \cdots + k_i = k$. We can obtain a fence from such a partition by imposing an order on it. Let S be the multiset of integers corresponding to an integer partition of k, and let $l = |S|$. Now, we can create a fence $F \in \mathscr{F}(k,l)$ from this partition by fixing $F = (k_1, \ldots, k_l)$ where $k_i \in S (1 \leq i \leq l)$.

[2] See Appendix A for more information about the *percy* library.

Note that S is a unique partition of k. However, F may not be the only fence corresponding to this partition. To see why, let π be a permutation of l. Then, the fence $F' = F_\pi = (k_{\pi(1)}, \ldots, k_{\pi(n)})$ is also a fence in $F(k, l)$.

Thus, to generate all fences in $\mathscr{F}_k$, we have to do the following:

- Generate all integer partitions S of k.
- For all such S, generate all permutations π_S.

In practice, we are often not interested in enumerating all 2^{k-1} fences in $\mathscr{F}_k$. Instead, we are often satisfied once we obtain a fence that our synthesizer finds a solution with. All of this suggest the lazy fence generating algorithm in Algorithm 4. The algorithm presented here is a coroutine that may be called repeatedly and yields all fences in $\mathscr{F}_k$ until exhausted. The algorithm is constructed by composing standard integer partitioning and permutation algorithms. In our implementation we use a lazy adaptation of the integer partition algorithm from Knuth [69, page 392], who attributes it to Hindenburg [63]. For the permutations we use an algorithm from the C++ standard library.

Algorithm 4 is an efficient procedure that may be function in the inner loop of a synthesis algorithm. For example, we can generate set $\{\mathscr{F}_k \mid k \leq 10\}$ in 0.097 seconds. On top of this basic procedure we can also build more sophisticated algorithms, such as algorithms that filter out any fences that are unnecessary for a specific synthesis task. We discuss such methods in Section 3.6.

3.5.2 Recursive Backtracking Method

This simple, but efficient, fence-generating method depends on the notion of a *node budget*. Suppose we want to generate the fence family $F(k, l)$ (i.e. all fences of k nodes and l levels, $k \geq l$). By definition of fences, we must have at least one node on each level. Hence, there remains a budget of $r = k - l$ nodes that we must distribute over the l levels. Each unique distribution corresponds to a valid fence. Thus, in order to generate all fences, it suffices to spend the budget of r nodes across the levels in all possible ways. Algorithm 5 shows how we can achieve this using a recursive backtracking approach. The F variable represents the current state of the fence being generated. It is a simple array of size l. The value stored at $F[i]$ represents how many nodes have distributed to level i. Note that $F[i] \geq 1$. The variable `budget` is initialized to r and tracks how many nodes we can still spend at a given time. In the base case of the recursion, this budget is zero. In that case there are no more nodes to spend and we simply yield the current state of the fence F. Afterwards, we backtrack to the previous state of the recursion to generate any remaining distributions. If we are not in the base case,

Algorithm 4 An algorithm to generate all fences in $\mathscr{F}_k$.

```
function GENERATEFENCES(k)
    while true do
        S ← NextPartition(k)
        if S ≠ ∅ then
            l ← |S|
            while true do
                π ← NextPermutation(S)
                if π ≠ ∅ then
                    F ← EmptyFence(l)
                    for i ← 0; i < l; i++ do
                        F[i] ← S[π(i)]
                    end for
                    yield F
                else
                    break
                end if
            end while
        else
            yield ∅
        end if
    end while
end function
```

there are nodes remaining to be spent. The variable `level` keeps an index which tracks what level of the fence we are currently determining our budget for. For example, if `level == 2`, then we are deciding how many nodes of our budget to spend on the second level of the fence. There is one exception here. If we are on level l, we *must* spend our entire remaining budget to obtain a valid fence in $F(k,l)$. If we do not, we would generate a fence $F'(k',l)$ where $k' < k$. We recursively generate all possible budgets from this point, backtracking after having done so. In the end, this procedure obviously generates all possible ways to spend an r-node budget.

Algorithm 5 A recursive backtracking algorithm to generate all fences in $F(k,l)$.

```
function GENERATEFENCES(k, l)
    if budget == 0 then
        yield F
        backtrack()
    else
        start-budget ← 0
        if level == l then
            start-budget ← budget
        end if
        for i = start-budget; i <= budget; i++ do
            budget ← budget - i
            F[level] ← i
            level ← level + 1
            GenerateFences(k, l)
        end for
        backtrack()
    end if
end function
```

3.6 Exact Synthesis Using Fences

We have seen how fences correspond to families of DAG topologies, investigated some of their theoretical properties, and presented a fence generating algorithm. In this section we consider how to use fences to accelerate exact synthesis by using them to provide additional constraints in the SAT formulation. To do so, let us first look at some connections between fences and Boolean chains.

Consider a fence $F = (\lambda_1, \ldots, \lambda_l)$. Let $G = (V, E)$ be a DAG, and let $\tau(v) : V \to \mathbb{N}$ be the function that assigns each vertex from G to its level. Let $\tau_i = |\{v \mid \tau(v) = i\}|$. We say that G *satisfies* F if and only if $|\lambda_i| = \tau_i$. In other words, a DAG satisfies the topological constraints of a fence if its distribution of nodes across levels is the same. We say that a Boolean chain satisfies F if its underlying DAG structure satisfies F. We consider the primary inputs of the chain to have

level 0, and do not consider them in satisfying F.

For example, consider the fence $F = (\lambda_1, \lambda_2) \in F(4,2)$ highlighted in Figure 3.3(a). We have numbered its nodes to make them easier to distinguish. Intuitively, only DAGs with two nodes on the first level and two nodes on the second level satisfy F. For example, Figure 3.3(b) is a 2-input operator Boolean chain satisfying the constraints from F. Similarly Figure 3.3(c) is a 3-input Boolean chain that satisfies F. However, Figure 3.3(d) shows a chain that is invalid for F. It violates the constraint that the step corresponding to fence node 4 be on level 2.

Observe that the topology constraints captured by fences are independent of number of inputs, or operator fanin. This is desirable, as it implies that the same fence generator can be used as the basis for synthesis of generalized Boolean chains and functions of arbitrary input size.

Now consider again the arbitrary fence $F = (\lambda_1, \ldots, \lambda_l) \in F(k,l)$. Suppose we wish to synthesize a Boolean chain that satisfies F. We know that it must be a k-step chain. We assign step x_i to level t by setting

$$\tau(x_i) = t \Leftrightarrow t = \min_{t'} i \leq \sum_{j=0}^{t'} |\lambda_j|.$$

where $|\lambda_0| = n$, the number of primary inputs.

Note that if $\tau(x_i) = t$, then step x_i must, by definition, have at least one fanin on level $t-1$. Thus, the fence constrains not only the distribution of nodes across levels, but also the fanin relations between nodes. Due to this level constraint, in the SAT formulation the selection variable s_{ijk} may never be true if $\tau(k) < t-1$, for any $i < k$. Let k' and k'' be the smallest and largest indices such that $\tau(x_{k'}) = t-1$ and $\tau(x_{k''}) = t-1$, respectively. A simple way to express the constraints imposed by the fence is by adding, for each step x_i, the clause $\bigvee_{k=k'}^{k''} s_{ijk} (j < k)$. In that way, we ensure that each step has at least one fanin from a level directly below. This approach is similar to the way that colexicographic or other symmetry-breaking clauses are added in [70]. However, we can do better. As none of the variables outside of $\{s_{ijk} \mid k' \leq k \leq k''\}$ may be true, we do not need to include them in our SAT formula at all. Thus, with fence we can significantly reduce both the number of variables and clauses in our SAT instances.

To implement exact synthesis with topological constraints we can then proceed as follows: (i) Generate a new fence using some fence-generating algorithm. (ii) Using the constraints implied by the fence, generate a reduced SAT formula. We use a set of clauses analogous to the one described in Section 2.2. However, we exclude any variables or clauses that are rendered unnecessary due to the fence constraints, obtaining a simpler SAT formula. (iii) If the formula is satisfiable, we are done. (iv) Otherwise, go to (i). If we incrementally increase the size of the fences that are generated this procedure is guaranteed to find a size-optimum chain. Thus, we extend the conventional exact synthesis algorithm, while decomposing the search space using

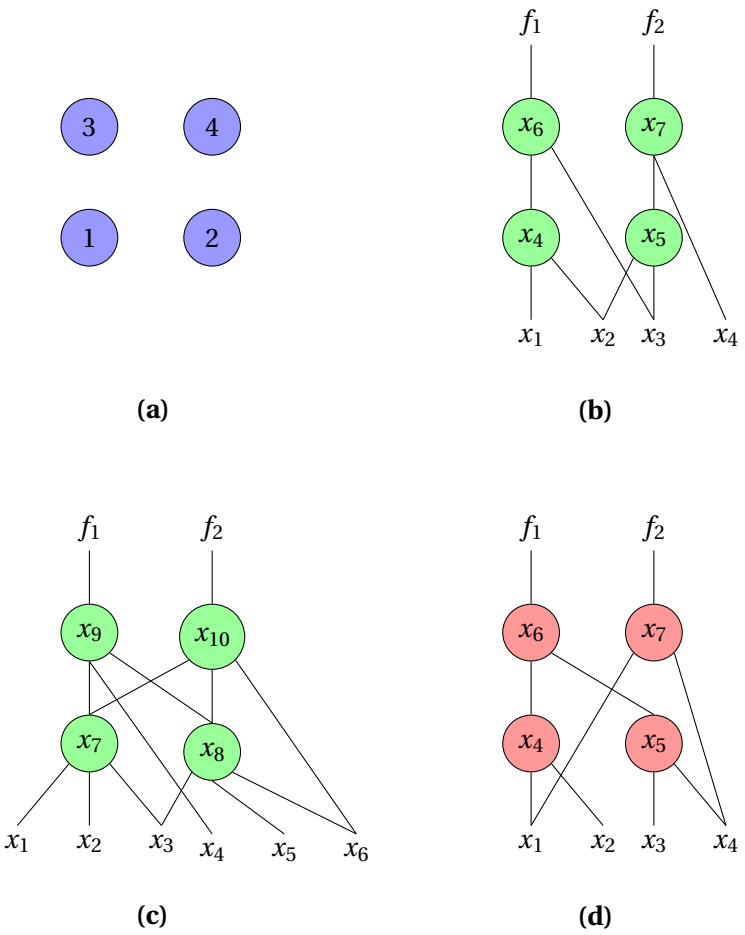


Figure 3.3 – The fence F in (a) corresponds to a set of possible DAG topologies and can thus be used to constrain the SAT solver's search. For instance, Figure (b) and Figure (c) satisfy the constraints from F. Figure (d) does not. Each node on level λ must have at least one fanin from level $\lambda - 1$; this follows by definition of levels.

families of graph topologies. Recall that in Section 3.4 we derived the total number of fences of k nodes. Given an upper bound on the number of nodes to realize a function, we therefore also have an upper bound on the number of decomposed exact synthesis instances we have to solve.

3.7 Fence vs. Conventional Encodings

To evaluate the performance of our proposed approach, we measure the runtimes of different exact synthesis encodings on the following collections of Boolean functions:

- *NPN4:* All 222 4-input NPN classes [64].
- *FDSD6:* 1000 fully-DSD decomposable 6-input functions that occur frequently in practical synthesis and technology mapping applications [89].
- *PDSD6:* 1000 common 6-input *partially*-DSD functions.
- *FDSD8:* 100 fully-DSD decomposable 8-input functions.
- *PDSD8:* 100 partially-DSD decomposable 8-input.

We compare three different encodings to synthesize 2-input operator chains for these sets of functions:

1. *SSV:* A baseline implementation of the SSV encoding described in Section 2.2. We enable all symmetry breaks described there, as we experimentally found that this works best for the synthesis of 2-input operator chains.
2. *Fence:* Our proposed algorithm based on fence enumeration and the use of additional topological constraints.
3. *Partial DAGs:* Our algorithm based on partial DAGs.

Table 3.2 lists the results. For each approach three values are listed: i) the mean solving time (*mean*) in milliseconds, ii) the number of instances that could not be solved in under three minutes (*#t/o*), and iii) the number of instances that were successfully solved within the timeout limit (*#ok*). Note that the number of solved instances is the most important metric here, as it captures in essence how practical an algorithm is. Given a bound on runtime, we obviously prefer the algorithm that can solve the most problems within that bound. A similar metric is commonly used in SAT solver competitions.

Table 3.2 – Comparing fence- and partial DAG-based synthesis to conventional state-of-the-art encodings. All runtimes in ms.

Benchmark	SSV			Fence			Partial DAG		
	mean	#timeouts	#ok	mean	#timeouts	#ok	mean	#timeouts	#ok
NPN4	225.46	0	222	216.69	0	222	75.40	0	222
FDSD6	69.00	0	1,000	29.61	0	1,000	82.41	0	1,000
PDSD6	43,453.33	256	744	20,707.11	128	872	3,613.25	5	995
FDSD8	5,583.13	0	100	2,688.51	0	100	31,379.47	0	100
PDSD8	150,533.31	42	58	100,871.79	11	89	131,625.42	84	16

The results show that using topological structure enumeration can significantly improve the solving time, as well as the number of solved instances. For *NPN4*, our fence-based algorithm is more than **19%** faster than our baseline implementation. All algorithms find the solutions for all problem instances. For *FDSD6*, *Fence* is **2x** faster than *SSV*. Again, there are no timeouts. For *PDSD6*, *Fence* is also **2x** faster than *SSV* and we also have **2x** fewer timeouts. The same observation can be made for the 8-input function sets. For *FDSD8*, *Fence* is again **2x** faster than *SSV*. Finally, for *PDSD8*, *Fence* is **63.43%** faster than *SSV*. Again, fence-based synthesis has fewer timeouts. In fact, the table shows that it dominates SSV with respect to the number of solved instances. In summary, we see that the gains from using topological constraints can be substantial.

3.8 Synthesis With Partial DAGs

Here, we compare synthesis based on partial DAGs to fence-based synthesis and conventional encodings. First, we apply partial DAG synthesis on the benchmarks described in Section 3.7. Table 3.2 contains the results. Partial DAGs allow us to improve runtimes on the NPN4 and PDSD6 benchmarks. On NPN4, partial DAGs obtain a runtime reduction of **3x** over both SSV and Fences. On PDSD6, the runtime reductions are **12x** and **5.5x**, respectively. Moreover, on the PDSD6 benchmark, they reduce the number of timeouts by 251 and 123 as compared to SSV and Fences, synthesizing all but 5 of the functions in under three minutes. Partial DAGs perform less well than SSV particularly on the FDSD8 and PDSD8 benchmarks. We conjecture that this is caused by the larger combinational complexity of the functions in those benchmarks. This forces partial DAG synthesis to try more topologies, thus slowing it down. However, we believe that our filtering methods can likely still be improved to further reduce the number of potential remedies.

In our next experiment, we compare SSV, fence-based, and partial DAG-based synthesis on a hard benchmark set. We sample 500 random 5-input functions, and try to synthesize optimum 2-input operator chains. Note that the majority of 5-input functions are hard, in that they require a large number of gates to implement [69]. In fact, it is true in general that most functions are random, and that random functions require exponentially many gates [112]. In

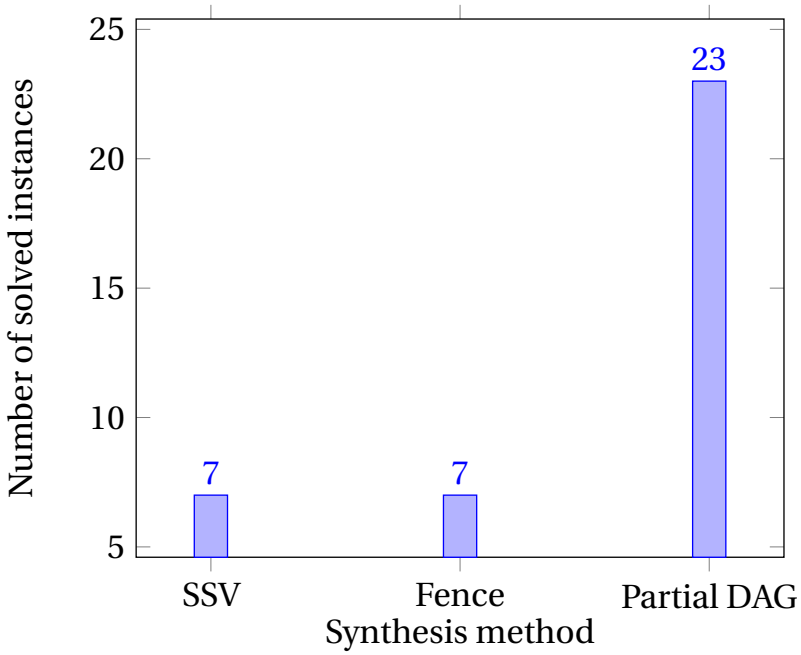


Figure 3.4 – Shows, for a set of 500 hard benchmarks, the number of successfully synthesized chains within the 1 minute timeout.

this experiment, we see how many functions these different methods can synthesize, setting a timeout at one minute. Figure 3.4 shows the results. We see that synthesis based on partial DAGs is able to synthesize more than **3x** as many functions in under one minute of runtime. We conclude that both fences and partial DAGs can unlock significant runtime improvements and can both be used to solve more problem instances, although the domains on which they are best used may be different.

3.9 Topology-Based Parallel Exact Synthesis

In this section, we outline and evaluate a parallel exact synthesis architecture based on topology families. We do not assume anything about the type of topology family. They may be fences, partial DAGs, or some other kind of topologies.

Suppose we are given a function f to synthesize. We can then produce a stream of topologies that may be used as a basis for f, using algorithms such as those described in Section 3.5. In this scenario it will be useful to consider the stream as a queue Q. We do not know in advance which topology can implement f. Therefore, the single-threaded algorithms described in Sections 3.6 and 3.8 sequentially pop topologies out of Q until they find one that applies. Now suppose we have n threads, all of which have access to Q. They can all pop topologies out of

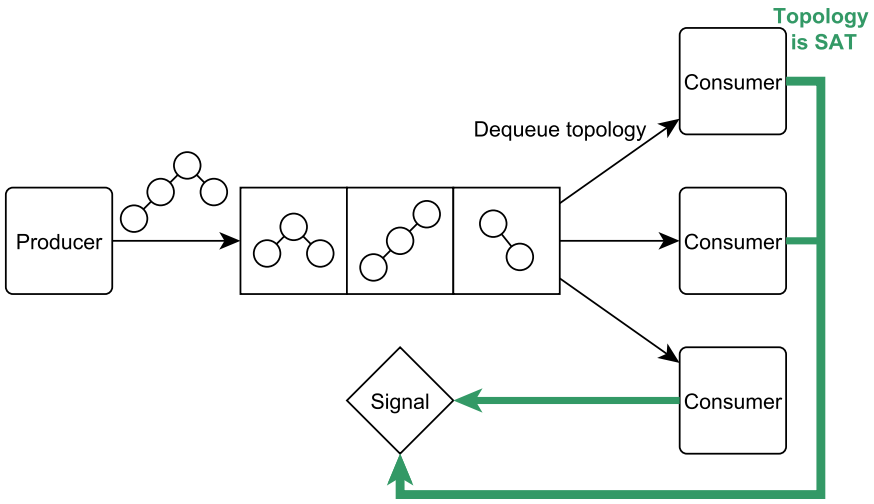


Figure 3.5 – Shows how topology information may be used to create an embarrassingly parallel exact synthesis pipeline.

Q until one of them finds a topology that works. As soon as a solution is found by thread t it can signal the other threads to stop working. In fact, the situation is slightly more nuanced. To guarantee a minimum solution, threads t' that are looking for solutions with fewer gates than t should not be stopped. Alternatively, we may stage the generation of topologies, first generating all topologies with one gate, then those with two gates, and so on. Generating stages in sequence, we can stop as soon as the first thread in a stage finds a solution. This second approach was used in our experiments here. This algorithm is embarrassingly parallel, as there are no dependencies between threads, and there is no communication required except for the signal that a solution has been found. See Figure 3.5 for an illustration of this parallel synthesis architecture.

3.10 Topology-Based vs. Generic Parallelism

The architecture we describe above is one of many possible approaches to parallel SAT-based exact synthesis. Another is to use a generic parallel SAT solver to solve the CNF formulas generated by some encoding. However, we conjecture that such an approach is suboptimal, as such a solver is domain independent. To verify this hypothesis, we synthesize 2-input operator chains for a set of 1000 5-input functions, using two different parallel synthesis approaches. The first uses the SSV encoding, with a parallel SAT solver backend. We use Glucose-Syrup MultiSolvers, which won gold in the parallel track of the 2017 SAT competition [12, 47]. The second uses our proposed parallel architecture, with partial DAGs as topology families. Each thread is assigned its own single-threaded SAT solver. We use the bsat solver, taken from

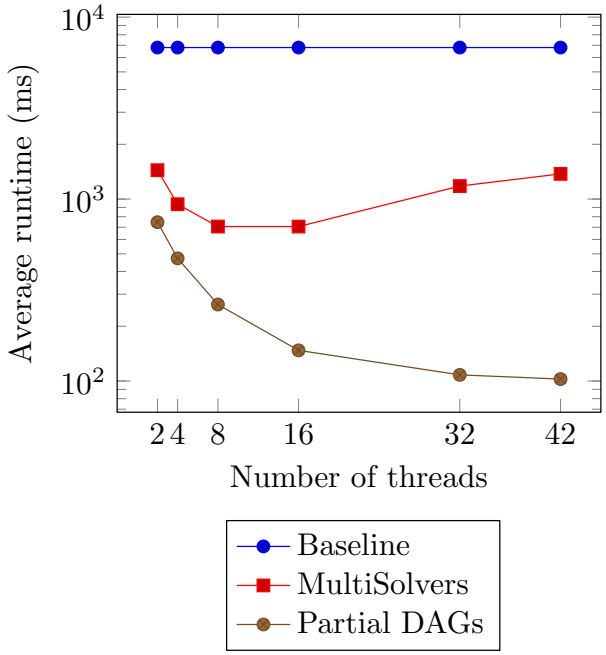


Figure 3.6 – A comparison between our domain-specific parallelism and a generic parallel SAT backend.

ABC [23]. Figure 3.6 contains the results. It also shows, as a baseline, the single-threaded performance of the bsat solver using the SSV encoding.

The results show that the MultiSolvers and partial DAG implementations are up to **9.5x** and **68x** faster than the single-thread baseline, respectively. The partial DAG implementation is up to **7x** faster than the best MultiSolvers configuration. Moreover, we see better scaling properties. The performance of partial DAG synthesis roughly doubles each time we double the number of threads. We do not see the same behavior using the MultiSolvers backend. In fact, its performance degrades after adding more than 16 threads. This is likely caused by increased thread contention as well a higher memory overhead as compared to our partial DAG implementation.

Interestingly, our implementation achieves a speedup of **68x** as compared to the single-thread baseline, even though it uses at most 42 threads. In other words, it obtains a super-linear speedup. To see how this is possible, consider Figure 3.7. It shows two topologies, F_1 and F_2, where F_2 can be used to synthesize a function, but F_1 cannot. Synthesizing sequentially, we must solve an UNSAT formula before a SAT one, which takes time $t_1 + t_2$. In a 2-threaded scenario, we can stop after $t_1 < \frac{t_1+t_2}{2}$ time, thus achieving a super-linear speedup.

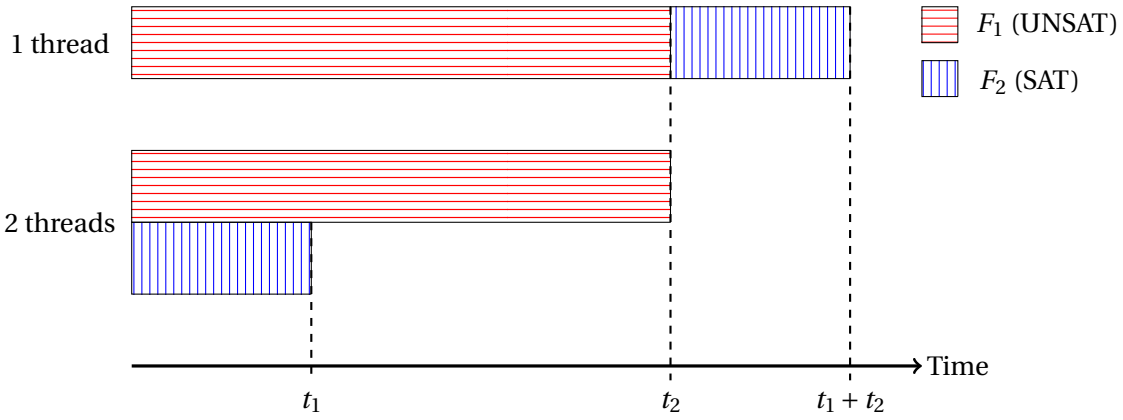


Figure 3.7 – Illustration of the super-linear speedup achievable by topology-based parallel synthesis.

3.11 Majority-7 Decomposition

Two major applications of exact synthesis are synthesis with novel logic primitives and finding new upper bounds for classes of circuits. Our second experiment in this section considers both of these objectives. It concerns the decomposition of majority-n functions. Recall that the majority-n function is defined as

$$\langle x_1 \dots x_n \rangle = \left[x_1 + \cdots + x_n > \frac{n-1}{2} \right] \quad (n \text{ odd}).$$

One often wants to find a decomposition of majority-n functions into majority-3 operations, as this is an important task in majority-based logic synthesis. This has applications in both classical logic synthesis as well as synthesis for emerging technologies [135]. Moreover, upper bounds for small circuits can help us find better theoretical upper bounds for larger ones [76]. Therefore, in this experiment we decompose the majority-7 function into an optimum network of majority-3 operators. We use the same parallel exact synthesis architecture as before, but this time using fences as the topology families. To show the impact of parallelism we attempt this decomposition with increasing numbers of threads. We compare against a conventional synthesis method that is based on an extension of the SSV encoding. The results can be found in Fig 3.8. In this figure, F/x refers to fence-based synthesis with x threads. The conventional approach requires 20,745ms. The single-threaded fence-based approach is **11%** faster, showing again the impact that topology-based synthesis can have even in the single-threaded case. With 2 threads, the fence-based synthesis is about **4x** faster. This is another example topology-based multi-threading unlocking super-linear speedups. Moreover, as we double the number of threads, synthesis time is cut approximately in half until we reach 16 gates. As we increase to 32 threads, runtime still decrease, but not as significantly. Finally,

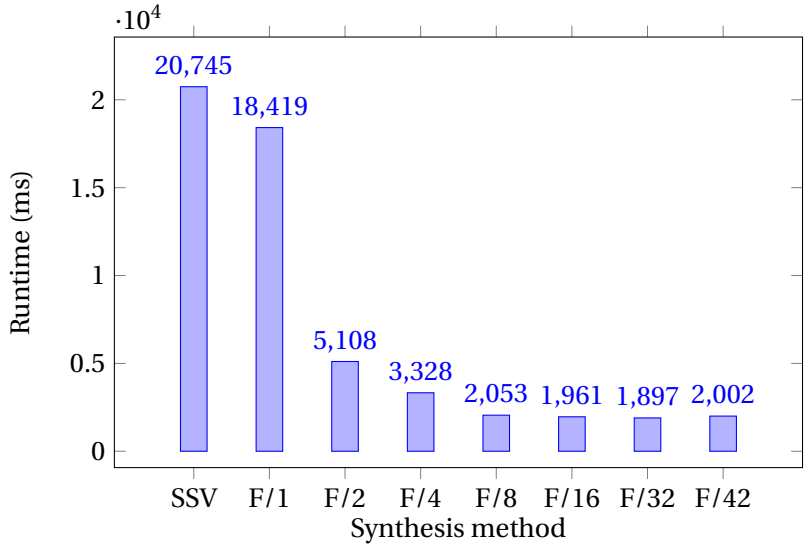


Figure 3.8 – Comparison of majority-7 decomposition between the best SSV encoding and a fence-based encoding with an increasing number of threads.

when go to 42 threads, we slightly degrade performance. We conjecture that the added cost of creating more threads outweighs the additional throughput they provide. The best runtime, 1897ms, is achieved by 32 threads. Thus, we achieve a runtime reduction of more than **10x**.

3.12 Summary

In this chapter, we have shown how the conventional unstructured SAT-based exact synthesis introduced in Chapter 2 can be improved by the use of DAG topology information. We have introduced the concept of DAG topology families and their relation to the synthesis of Boolean chains. We have given two examples of such families – Boolean fences and partial DAGs – and shown how they both can be advantageous when compared to conventional synthesis. Finally, we have shown how topology-based synthesis can be used as the basis for parallel exact synthesis. This breaks a long-standing barrier, as logic synthesis algorithms have traditionally proven hard to parallelize. We have demonstrated that parallel exact synthesis can enable substantial performance improvements over sequential synthesis, unlocking even super-linear speedups. This chapter concludes the core algorithms part of this thesis. The next starts the applications part.

Applications Part II

4 Function Classification

As we have demonstrated in Chapters 2 and 3, SAT solvers have become efficient tools for synthesizing optimum Boolean circuits. In this chapter, we examine a first, theoretical, application of SAT-based exact synthesis, by showing how it may be used as a method for *classifying* Boolean functions. As opposed to previous classification methods, ours may be easily parallelized, which we use to obtain a speedup of approximately 48x. Combining our method with NPN canonization, we find, for the first time, the minimum-size chains for all 4- and 5-input functions in terms of 3-input Boolean operators.

The remainder of this chapter is organized as follows. First, in Section 4.1, we define the problem statement of this chapter more formally, and provide pointers to related work. Next, in Section 4.2, we revisit the preliminaries of NPN canonization, as it is an essential part of our classification algorithm. In Section 4.3, we give a detailed description on our SAT-based classification method. In Section 4.3.1, we show how the conventional NPN classification algorithm can be adapted to more efficiently find all relevant NPN classes. In Section 4.3.2 we describe our exact synthesis method to compute $C(f)$ for all functions and NPN classes. Section 4.4 contains the experimental results. Finally, we conclude and draw a future outlook in Section 4.5.

4.1 Introduction

The main goal of this chapter is to use exact synthesis as a tool for classifying functions in terms of their *combinational complexity*. Given a function f, its combinational complexity $C(f)$ is defined as the size of the minimum-sized chain that computes f [69]. As such, combinational complexity is a natural measure of the inherent complexity of f. Surely, functions that are harder to compute require more steps. Thus, classifying functions in terms of $C(f)$ gives us some insight into the "hardness distribution" of Boolean function. Moreover, synthesizing

chains using different step operator sizes gives insight into the relation between operator size and combinational complexity. For instance, one might suspect that computing f with larger operators requires fewer steps, since the operators are more expressive.

There are several research directions that address questions about combinational complexity in different ways. We may consider these directions as consisting of three different categories. The first category is concerned with finding sets of primitives such that the complexity of all Boolean functions f satisfies some upper bound (see, e.g., [49, 60, 61]). The second is concerned with finding complex Boolean functions that satisfy a lower bound for some given set of primitives (see, e.g., [21, 105, 121]). Finally, the third is concerned with finding exact numbers for the combinational complexity given a subset of Boolean functions and a set of primitives P (see, e.g., [69, 129, 122]). The results we present in this chapter falls into the third category. This work is of theoretical interest, as it gives us more concrete information about the complexity distribution of Boolean functions. Moreover, having exact numbers for the combinational complexity of some small functions can help us find tighter upper bounds for larger functions [76].

Besides being of theoretical academic interest, there is a more practical side to finding and classifying Boolean functions in terms of minimum-size chains. For example, as we will see in the next chapter, such chains can be used in applications such as logic optimization and technology mapping. Moreover, the exploration of minimum-size chains is motivated by emerging and existing technologies. In recent years, different nanotechnologies have been implementing more powerful devices that go beyond the capabilities of traditional NAND/NOR gates [143, 78, 43]. These devices implement more expressive operators, such as 3-input majority or minority functions. More traditionally, gates such as multiplexers also correspond to 3-input operators. Hence, finding optimum chains based on 3-input operators can help in the design of circuits based on such technologies.

In [69], Knuth shows how to compute the combinational complexity of all 4- and 5-input Boolean functions composed of all 2-input Boolean operators. He provides the exact numbers for the combinational complexity for all 222 NPN classes of the 4-input functions. He computes these numbers by efficiently by enumerating all Boolean chains until some chain for each function has been encountered. In Table 4.1 we show, for reference, his results for the combinational complexity for all 4-input functions. A more sophisticated algorithm is required to find exact numbers for the combinational complexity for all 616,126 NPN classes of the 5-input functions. Yet, "thanks to a bit of good luck" (as stated in [69]) and the computer program `BOOLCHAINS`,[1] it was possible to find the numbers presented in Table 4.2. However, significant modifications to the main algorithm were required to find the numbers for some of these classes. Certain classes were handled as special cases. The vast majority of computation

[1] http://www-cs-faculty.stanford.edu/~uno/programs/boolchains.tgz

Table 4.1 – Combinational complexity of all 4-input functions using 2-input operators [69]

$C(f)$	Classes	Functions
0	2	10
1	2	60
2	5	456
3	20	2474
4	34	10624
5	75	24184
6	72	25008
7	12	2720

Table 4.2 – Combinational complexity of all 5-input functions using 2-input operators [69]

$C(f)$	Classes	Functions
0	2	12
1	2	100
2	5	1140
3	20	11570
4	93	109826
5	389	995240
6	1988	8430800
7	11382	63401728
8	60713	383877392
9	221541	1519125536
10	293455	2123645248
11	26535	195366784
12	1	1920

time was spent finding the 11-step chains for their 6 corresponding NPN classes and the 12-step chain for its single corresponding NPN class.

In this chapter, we conduct a modified version of Knuth's experiment, with two major differences. First, in our version of the experiment, we synthesize 3-input operator Boolean chains, thus allowing the full set of 3-input operators to be used as logic primitives. Second, rather than using his chain enumeration method, we propose to use SAT-based exact synthesis, applying the techniques described in Chapter 2. Our method removes the requirement to explicitly enumerate all Boolean chains. Furthermore, it does not require any modifications to handle special cases. Finally, it is easily parallelized.

Following Knuth, we take advantage of the property that all functions which are equivalent up

(a) (b)

Figure 4.1 – An example of two different functions that are P-equivalent. The circuit in (a) can be made equivalent to the one in (b) by permuting the inputs.

(a) (b)

Figure 4.2 – An example of two different functions that are NPN-equivalent. The circuit in (a) can be made equivalent to the one in (b) by negating its output and permuting the inputs.

to input negation, input permutation, and output negation (i.e. NPN equivalent, [50]) have the same combinational complexity. This allows us to consider a subset of 222 and 616,126 functions instead of 65,536 and 4,294,967,296 functions for 4 and 5 inputs, respectively.

4.2 NPN Canonization

Two functions are P-equivalent if they are equivalent up to permutation of their inputs. For example the functions $f = a \cdot (\bar{b} + c)$ and $g = c \cdot (\bar{a} + b)$ are P-equivalent, since we can make them equal by swapping the inputs a, b, and c. This is illustrated by Figure 4.1. NPN equivalence is a generalization of P equivalence. We say that two functions are NPN-equivalent if they are equivalent up to permutation of their inputs *and* negation of their inputs and output [58, 65]. For example, the functions $h = a \cdot b + c$ and $i = \bar{c} \cdot \bar{a} + \bar{b} \cdot \bar{a}$ are NPN-equivalent, since h can be made equivalent to i by negating its output and swapping inputs a and c. See Figure 4.2 for an illustration.

NPN-equivalence is an equivalence relation. Indeed, it can be easily checked to be reflexive, symmetric, and transitive. Thus, the space of Boolean functions is partitioned into disjoint sets of NPN-equivalent functions which we call NPN-classes. When two functions f and g are NPN-equivalent, we say that they are members of the same NPN class, and we write $f \sim g$. We

use $[f]$ to denote the NPN-class, and define it as

$$[f] = \{h \in \mathbb{B}^{2^n} \mid h \sim f\}.$$

We pick one function $\hat{f} \in [f]$ to be the *equivalence class representative*. We say that $\hat{f}$ is the *canonical* representative of $[f]$. Hence, the term NPN canonization. We often use the terms NPN canonization and NPN classification interchangeably, as finding the NPN classes and representatives are closely related tasks. Typically, the function $f \in [f]$ whose truth table corresponds the smallest integer value is chosen to be the equivalence class representative.

An important motivation for NPN classification is simply the large number of Boolean functions. As we know, the number of n-input single-output Boolean functions is 2^{2^n}. Hence, any method that needs to access all individual functions quickly becomes intractable, and we often want to avoid doing so. NPN classification enables this, by grouping Boolean functions into classes. The number of NPN classes of n-input functions is much smaller than the total number of functions. Indeed, Table 4.3 shows that it is many orders of magnitude smaller than the number of functions. At the time of writing, no closed-form solution for the number of n-input NPN classes is known. There are ways compute these numbers, but they are computationally intensive, and quite complicated to describe. They go back to methods developed in Harrison's PhD thesis, in which he improves on earlier work by Elpas and Ninomyia [59]. Understanding these methods requires a significant amount of background knowledge and we will not attempt to describe them here. We refer the interested reader to [59, 58, 128]. Importantly, as a result of the NPN equivalence relation, many useful properties of NPN classes are the same for all members within a class. In other words, if we want to compute some property of the functions in $[f]$, it often suffices to only compute it for $\hat{f}$. Therefore, NPN canonization is a useful tool for the study of Boolean functions. Besides the use of NPN classification in this chapter, it has applications ranging from Boolean matching to logic rewriting and exact synthesis [64, 93, 52]. Efficient exact and heuristic algorithms for NPN classification have been developed over the years [64, 107, 130].

Since we want to classify functions in terms of $C(f)$, we can use the properties of NPN equivalence to our advantage. The equivalence relation depends only on permutations and negations. Therefore, it does not affect the size of Boolean chains. In other words, the minimum-size Boolean chain for any $f \in [f]$ can be derived from chain for $\hat{f}$, simply by applying the proper permutations and negations (i.e. moving around the chain inputs). More formally, if $g \in [f] \Rightarrow C(\hat{f}) = C(g)$. Hence, to find $C(f)$ for all n-input functions we do not need to examine all functions. Instead we can find $C(\hat{f})$ only for the NPN class representatives. This is preferable, since the number of NPN classes is significantly smaller than the number of functions. Table 4.3 lists the number of functions and classes for up to 8 inputs to illustrate how great this difference is. We can see that, although the number of NPN classes still grows

Table 4.3 – Comparing the number of n-input functions and NPN classes. Numbers of NPN classes taken from [127]. We write the numbers for $n = 8$ in scientific notation, as they would not fit on the page otherwise.

n	Number of functions	Number of NPN classes
0	1	1
1	4	2
2	16	4
3	256	14
4	65,536	222
5	4,294,967,296	616,126
6	18,446,744,073,709,551,616	200,253,952,527,184
7	340,282,366,920,938,463,463,374,607,431,768,211,456	263,735,716,028,826,576,482,466,871,188,128
8	1.158×10^{77}	5.609×10^{69}

rapidly with n, it is orders of magnitude smaller than the number of functions.

4.3 Classification Method

Our main goal in this chapter is to classify all 4- and 5-input functions in terms their combinational complexity with 3-input operators. We present a method that achieves this goal using a combination of NPN classification and exact synthesis. Our method can be divided into two parts: (i) finding all 4/5-input NPN classes, and (ii) using exact synthesis to find the combinational complexity for all 4- and 5-input functions and NPN classes. Roughly speaking, we use part (i) to reduce the computational work required in part (ii).

4.3.1 Finding All NPN Classes

In order to find the representative $\hat{f}$ for a given function f, one needs to visit all functions in $[f]$ to select the smallest one. If f has no helpful properties—such as symmetries in the inputs (see, e.g., [107, 1])—one needs to apply all possible combinations of 2^n input negations and $n!$ input permutations for both f and $\bar{f}$. In order to reduce the effort, we can use a smart ordering in which all these transformations are applied. We can use gray code enumeration to invert inputs, thereby flipping only one bit at a time. In a similar way we can use *plain changes* (see, e.g., [101]) to visit all permutations by swapping two adjacent inputs at each time. It is possible to combine both concept in an enumeration algorithm that visits all *signed permutations*, i.e., permutations in which elements can be complemented [69]. In combining the steps outlined here, we can construct an NPN canonization algorithm that is quite efficient. However, canonization still comes at a non-trivial computational cost. Therefore, we would like to reduce the number of times we are required to invoke the classification algorithm.

Recall that our goal is to find the NPN classes for all functions. A naive algorithm would be to simply iterate over all functions, computing and saving the class representative for each functions. However, such an algorithm would have to invoke the classification algorithm 2^{2^n} times. This already becomes impractical when we want to classify the 5-input functions. Fortunately, we can use the inner workings of the classification method to our advantage and avoid most of this computational work. All elements in $[f]$ are visited when computing $\hat{f}$. We can store this information to avoid any redundant classification efforts, by marking which functions are visited. The pseudocode for our classification algorithm is shown in Algorithm 6. Let us consider how it reduces the amount of work to be done. There are $2^{2^5} = 4,294,967,296$ single output 5-input functions. We initialize a map R that is indexed by the 5-input functions and initialize each of its 2^{2^5} elements to $\emptyset$ (null). Next, we essentially repeat the following steps:

1. We find the first f for which $R(f) = \emptyset$. This can be done efficiently by a simple linear scan of R. If no such f exists, we are done.
2. Compute $\hat{f}$ using a standard NPN canonization algorithm. While the classification algorithm visits the elements f' in $[f]$, set $R(f') \leftarrow (\hat{f}, \pi_{f'})$, where $\pi_{f'}$ is the signed permutation which transforms $\hat{f}$ to f'.
3. Go to step 1.

This loop reduces the number of invocations of the classification algorithm from 616,126, as it is now called exactly once for every NPN class. After the loop has finished, the image of R is the set of all function representatives. Moreover, we can emit $\hat{f}$ anytime we compute a canonical representative. Thus, while constructing the map R, we can construct, in parallel, a list L of all NPN classes. Finally, we can easily create another map N, which maps every class representative to a number representing the size of that class. Hence, after running this algorithm, we have three outputs: (i) a map R which can be used to map functions to their class representatives, and (ii) a list L of all NPN representatives, and (iii) a map N from the NPN classes to their sizes. In Section 4.3.2, we show how these can be used in our classification efforts.

4.3.2 Finding Minimum-Size Chains With Exact Synthesis

After efficiently finding the NPN classes L, we use exact synthesis to find the corresponding optimum Boolean chains. We use one of the conventional SAT-based methods introduced in Chapter 2.[2] To find all minimum-size chains, we simply apply our exact synthesis to every

[2] We use conventional here to mean not topology-based.

Algorithm 6 An algorithm which computes the following objects: (i) a list L of all n-input NPN classes, (ii) as a map R from the n-input functions to their respective classes, and (iii) a map N from the NPN classes to the *number* of functions in that class.

function COMPUTENPNDATA(n)
  $L \leftarrow []$
  $R \leftarrow NewMap(2^{2^n})$
  $N \leftarrow NewMap(0)$
  $InitializeMap(R, \emptyset)$
  for $i \leftarrow 0$; $i < 2^{2^n}$; $i \leftarrow i+1$ **do**
    if $R(i) \neq \emptyset$ **then**
      continue
    end if
    $f \leftarrow IntToFunction(i)$
    $(\hat{f}, [f]) \leftarrow Canonize(f)$
    for $f' \in [f]$ **do**
      $\pi_{f'} \leftarrow SignedPermutation(\hat{f}, f')$
      $R(f') \leftarrow (\hat{f}, \pi_{f'})$
      $N(\hat{f}) \leftarrow N(\hat{f}) + 1$
    end for
    $L \leftarrow L \circ \hat{f}$
  end for
  return (L, R, N)
end function

NPN class. As we have seen in Chapter 2, the use of 3-input operators significantly speeds up synthesis time. Moreover, using 3-input operators we uncover novel results: the minimum-size chains in terms of 2-input operators are known, but the minimum chains in terms of 3-input operators are not.

A key difference between our method and others is that ours is easily parallelized. Exact synthesis may be invoked in parallel on every NPN class we find, as there are no dependencies between invocations. Other methods would be significantly harder to parallelize. For example, enumeration based methods work by searching the space of possible circuit structures [69]. The search proceeds sequentially, yielding a sequence of optimum chains. This is not a process that is trivial to parallelize, as lower parts of the search space tree depend on choices made above.

Using our NPN classification method to find (L, R, N) greatly reduces the number of required exact synthesis invocations. However, at 616,126 5-input NPN classes, L may still be quite large. To reduce the runtime of our experiments, we run exact synthesis on each NPN class in L in parallel. Note that the type of parallelization we use here is not related to the topology-based techniques from Chapter 3. Rather, we may simply any number of worker threads in parallel to L. Each worker thread processes NPN classes independently of the others. Thus, this phase of our classification method consists of the parallel construction of the map S, which maps NPN classes to their optimum Boolean chains. Once this step has finished, we can easily use the L, R, N, and S to gather the classification statistics we require. Moreover, they can be used as the basis for a very fast, precomputed, exact synthesis algorithm. To see how, consider the pseudocode in Algorithm 7. Essentially, exact synthesis can now be performed through two map lookup operations and a signed permutations. Given a function f, we first map it to its NPN class, using R. We then map the NPN class to its optimum Boolean chain, chains, using S. Since we have also saved the corresponding signed permutation in R, we simply apply it to the chain, and we are done.

Algorithm 7 A fast exact synthesis algorithm, which finds the minimum-size chain for f using the precomputed objects R and S.

```
function PRECOMPUTEDSYNTHESIS(f, R, S)
    (f̂, π_f) ← R(f)
    c ← S(f̂)
    c' ← ApplySignedPermutation(π_f, c)
    return c'
end function
```

In some cases, it may be desirable to make small modifications to Algorithms 6 and 7. For instance, in the context of DAG-aware logic rewriting, it is advantageous to match multiple DAG structures with the same function [93]. Our method can be easily adapted to support this.

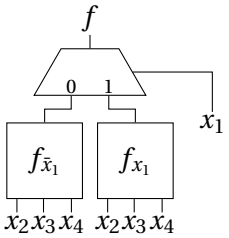


Figure 4.3 – We can implement any 4-input Boolean function by using at most three 3-input operators.

Algorithm 6 computes, and stores, only optimum Boolean chain per NPN class in S. Instead, we can enumerate a list of minimum chains, each with a different underlying DAG structure. This list is then stored in R. The rewriting algorithm can exploit this information by choosing, from this list, the chain whose structure matches best with that of the subject graph. By taking advantage of structure sharing between chain and subject graph, this may lead to increased size reductions. Indeed, such an adaptation of the method described in this chapter was used as the basis for the DAG-aware logic rewriting implementation in [111]

4.3.3 Synthesis Upper Bounds

Before conducting the actual experiment, it is instructive to consider some theoretical upper bounds on the minimum chain sizes we might expect to find. For both 4-input and 5-input functions we can find tight upper bounds on the size of minimum Boolean chains. The set 3-input operators includes the 2-to-1 multiplexer. We can use this operator to efficiently decompose functions and to find an upper bound. For example, we can write any 4-input Boolean function as $f(x_1, x_2, x_3, x_4) = \bar{x}_1 \cdot f(0, x_2, x_3, x_4) + x_1 \cdot f(1, x_2, x_3, x_4)$. This is known as Boole's expansion, and can be implemented by a 2-to-1 multiplexer. Note that the cofactors of f are 3-input functions. Consequently, they can both be implemented by a single 3-input operator. Figure 4.3 shows a sketch of this decomposition. Therefore, by using a multiplexer to do the initial expansion and two operators to implement the cofactors, we can implement any 4-input function with at most three 3-input operators. A similar argument can be used to show that we can implement any 5-input function using at most seven 3-input operators.

4.4 Experimental Results

In our experiments, we use a machine with 2 Intel Xeon E5-2680 v3 (Haswell) CPUs, each of which has 12 cores with support for 2 hyperthreads. For the computation of S we take full advantage of our hardware by using 48 worker threads. For exact synthesis we use the MSV encoding introduced in Chapter 2, with all symmetry breaks enabled.

Table 4.4 – Combinational complexity of all 4-input functions using 3-input operators

			Computation time (sec)		
$C(f)$	Classes	Functions	Avg.	Max.	Total
0	2	10	0.000	0.000	0.000
1	12	932	0.001	0.002	0.014
2	117	34,250	0.001	0.002	0.173
3	91	30,344	0.003	0.005	0.245

We first compute the statistics for all 4-input functions. The results of this experiment can be found in Table 4.4. It was constructed using the information in L, N, and S. The table shows, for each combinational complexity $C(f)$, the number of NPN classes and corresponding functions with that complexity. Further, it shows run time information. For each value of $C(f)$ we show the average, maximum, and total synthesis run time (in seconds) for all the NPN classes with that combinational complexity. The results show that our SAT-based synthesis algorithm is efficient: it never requires more than 0.005 seconds to synthesize any 4-input function. Furthermore, the results show that the upper bound we derived for 4-input functions in Section 4.3.2 is tight. There are exactly 91 NPN classes, corresponding to 30,344 functions, that cannot be implemented using fewer than 3 operators.

Next, we apply our method to finding the complexity distribution for all 5-input functions. The results can be found in Table 4.5. Interestingly, the upper bound we found in Section 4.3.2 was not tight in this case. Every 5-input function can be implemented by using at most 5 3-input operators.

As shown by Table 4.5, SAT-based exact synthesis again turns out to be an efficient method for finding the minimum chains. No function requires more than an hour to be synthesized, and the average synthesis time is just 8.938 seconds. This means that our method is able to find all minimum-size chains, without having to resort to different handling of special cases.

Despite the relative efficiency of SAT-based exact synthesis, the total sequential CPU time necessary to find *all* minimum-size chains is still $8.938 \times 616,126 = 5,506,943.478$ seconds, simply due to the large number of functions. By running on 48 threads in parallel the total wall clock time reduces significantly. We are able to synthesize all functions in approximately 1.5 days, thus taking full advantage of the embarrassingly parallel nature of our method. As mentioned above, other exact synthesis methods, such as those based on exhaustive enumeration of Boolean chains, are much harder to parallelize. Therefore, even if we assume faster average run times, they may still not be as practical.

Table 4.5 – Combinational complexity of all 5-input functions using 3-input operators

			Computation time (sec)		
$C(f)$	Classes	Functions	Avg.	Max.	Total
0	2	12	0.000	0.000	0.000
1	12	2,280	0.001	0.002	0.017
2	311	395,676	0.003	0.005	0.911
3	12,257	58,519,472	0.021	0.089	260.550
4	339,739	2,321,397,216	1.805	57.898	613,082.000
5	263,805	1,914,652,640	18.550	3,261.770	4,893,600.000

4.5 Summary

In this chapter, we have presented a new method for classifying all 4- and 5-input functions in terms of their combinational complexity with 3-input operators. Our method uses a combination of NPN classification and exact synthesis. We have shown that our exact synthesis implementation is efficient and that it is able to find all minimum-size chains without the need to handle any special cases differently. Moreover, it can be easily parallelized, which we use to enable an approximate 48x reduction in runtime. For the first time, we have presented the sizes of these minimum chains. Finally, since our method is based on a constructive synthesis method, it can be used not just to count the numbers of NPN classes and functions. With a slight modification it can be used as the basis for efficient exact synthesis algorithms, based on a two-layer precomputed index which maps functions to their NPN classes, and NPN classes to optimum Boolean chains. As we will see in the next chapter, such algorithms can then be applied to large-scale logic restructuring algorithms, such as logic rewriting.

5 Optimizing XOR-Majority Graphs

In practice, SAT-based exact synthesis is only applied to small functions, due to its high computational complexity. However, this does not make it irrelevant to the optimization of large functions. In this chapter, we will investigate an optimization method which uses exact synthesis to optimize large logic networks through a well-established technique known as *logic rewriting*. Specifically, we will apply exact synthesis to the optimization of XOR-Majority Graphs (XMGs). In Chapter 6 we will generalize the methods presented in this chapter to arbitrary Boolean networks.

The remainder of this chapter is organized as follows. In Section 5.1, we give an introduction to logic rewriting, provide pointers to previous work, and examine some of the limitations of previous methods. We also discuss some preliminary concepts, such as cut enumeration and LUT mapping, that are used in subsequent sections. In Section 5.5 we describe our generic optimization method, and Section 5.6 details a specific implementation of this method for the optimization of XMGs. Section 5.7 contains some experiments in which we evaluate our new method. Finally, we conclude and summarize our results in Section 5.8.

5.1 Introduction

Compared to two-level logic, exact optimization for multi-level logic networks has turned out to be more difficult due to its high computational complexity [77, 42, 44]. Conventionally, multi-level logic networks have been implemented as DAGs in which local node functions are represented by BDDs or *sum-of-product* expressions (i.e. DNF expressions). For such networks, Boolean and algebraic methods have been the driving force behind logic optimization algorithms [26]. Academic tools that implemented these algorithms famously include the MIS and SIS systems [25, 123]. Introduced by Brayton and McMullen [22], the algebraic optimization model treats Boolean expressions as regular polynomials over a field. Therefore,

the standard algebraic rules apply under this model. For instance, we may refactor the 3-input Boolean function $f = ax + bx$ as $x \cdot (a + b)$, since multiplication distributes over addition in polynomials. On the other hand, identities that rely on strictly Boolean properties, such as $x + \bar{x} = 1$, are no longer valid. Boolean optimization methods try to account for the limitations of the algebraic model by, as their name suggests, considering the full Boolean model. These methods are a collection of logic network transformations that use the additional degrees of freedom provided by so-called Boolean *don't care* conditions. Such conditions occur because of restrictions on network input as well as patterns that cannot occur due to network structure. Algorithms for the computation of don't care conditions go back some time, starting with work by Coudert et al. [39], Touati et al. [142], and Damiani and De Micheli [40]. More recently, with the rise of efficient SAT solvers, there has been increased interest in the development of SAT-based algorithms [95]. In this context SAT is not used as the basis for exact synthesis. Rather, such methods show that SAT can efficiently assist existing algebraic and Boolean algorithms, for example by using SAT to compute don't cares [94] or to improve technology mapping results [100, 119].

In recent years, we have seen a shift from complex heterogeneous logic representations to simpler homogeneous networks such as *And-Inverter Graphs* (AIGs) and *Majority-Inverter Graphs* (MIGs). Corresponding algebraic and Boolean optimization methods have been developed [93, 75, 7, 8, 9]. These simple data structures enable more efficient logic representation and optimization, requiring less memory and allowing better runtimes [23]. They also permit the implementation of fast logic rewriting algorithms, which improve the subject graph locally by replacing subnetworks with their precomputed optimized representations [93, 147]. In this chapter, we consider logic rewriting for another novel type of logic network: the *XOR-Majority Graph* (XMG).

5.2 Preliminaries

In this section, we go over some preliminaries that are essential for understanding our optimization method described in Section 5.5. They are also important factors in the more general optimization methodology described in Chapter 6. Therefore, they may be viewed as preliminaries for that chapter as well. They are self-contained, so the reader may feel free to skip or read them in any order.

5.2.1 Cut Enumeration

To optimize a large network with exact synthesis, we must first partition it into smaller subnetworks. If we make these subnetworks small enough, they will correspond to local functions for which exact synthesis is tractable. A common and efficient method for finding such sub-

networks is *cut enumeration*. As our proposed rewriting algorithm makes use of it, we briefly discuss its core concepts here.

Let N be a logic network. We do not make any assumptions on the particular logic network model here, such as whether it is homogeneous or heterogeneous. We only assume that N is a DAG. Let n be a node in N. A *cut* of n can be defined as a tuple (n, I), where I is a set of nodes of nodes in the transitive fan-in of n such that every path from a primary input to n passes through a node in I. See Figure 5.2 for an example. We say that a cut (n, I) is *redundant* if there is some $I' \subset I$ such that (n, I') is also a cut of n. A k-feasible cut of n is an irredundant cut (n, I) such that $|I| \leq k$. For any node n, we consider the *trivial cut* $(n, \{n\})$ to be a valid k-feasible cut. We refer to the elements in Ias the *leaves* of a cut.

Let X be the set of primary inputs of N, and let $\top$ and $\bot$ denote the Boolean constants. Suppose that m is a node in N, then inputs(m) is a set of node corresponding to the fan-in of m. We say that inputs$(x) = \emptyset$ for $x \in X$. We use $\Phi(m)$ to denote the set of k-feasible cuts for node m in N. We may define Φ recursively as:

$$
\begin{aligned}
&\Phi(\bot) = \emptyset \\
&\Phi(\top) = \emptyset \\
&\Phi(x) = \{(x, \{x\})\} && \text{for } x \in X \\
&\Phi(n) = \{(n, \{n\})\} \cup (\Phi(n_1) \otimes_n \cdots \otimes_n \Phi(n_m)) && \text{for } n \in N
\end{aligned}
$$

where inputs$(n) = \{n_1 \dots n_m\}$ and $\otimes_n$ is an an operation that gives an over-approximation of the k-feasible cuts of a node. It is defined as

$$A \otimes_n B = \{(n, a \cup b) \mid (n_1, a) \in A, (n_2, b) \in B, |a \cup b| \leq k\}$$

This characterization of Φ is due to [104] and [28]. As defined here, Φ may lead to the inclusion of some redundant cuts, but these can be easily filtered out during cut enumeration. For more details on efficient cut computation, we refer the interested reader to [97].

Using Φ as our computation method, the number of cuts for a node may be very large. Therefore, we often want to use some additional parameters l and p which bound the maximum cut size, and the maximum number of cuts we store for each node, respectively. This technique is referred to as priority cuts [98], as it selects the subset of all cuts with respect to some cost function, in our case the number of the cuts' leaves.

Algorithm 8 sketches the cut enumeration procedure that is used by our rewriting algorithm, both in this chapter as well as Chapter 6. It omits details on truth table computation and cut pruning based on functional dependence. The algorithm returns on termination a map from node n to a list of leaves $C(n)$ such that every pair (n, L) for $L \in C(n)$ is a cut of the Boolean

Algorithm 8 A cut enumeration algorithm. Returns a sorted list $C(n) = \{L_1, \ldots, L_q\}$ of cuts for every node n in N, such that $|L_i| \leq l$ and $|C(n)| \leq p$.

```
function CutEnumerate(Logic network N, cut size l, cut limit p)
    for each primary input n in N do
        C(n) ← {{n}}
    end for
    for each gate n in N in topological order do
        Let n_1, n_2, ..., n_m be the fanin nodes of n
        for each I_1 ∈ C(n_1), I_2 ∈ C(n_2), ..., I_m ∈ C(n_m) do
            I ← I_1 ∪ I_2 ∪ ··· ∪ I_m
            if |I| ≥ l then
                continue
            end if
            if ∃I' ∈ C(n) : I' ⊆ I then
                continue
            end if
            Remove all I' from C(n) for which I ⊂ I'
            Insert I into C(n) and keep C(n) sorted
        end for
    end for
    if |C(n)| > p then
        Remove the last |C(n)| − p elements from C(n)
    end if
    return C
end function
```

network. We sort the cuts in $C(n)$ by their size. In addition to basic cut function computation, we also compute the cut function's Boolean controllability don't cares, which are based on the local structure of the logic network. Finally, the returned cut sets are irredundant and do not contain two cuts (n, L_1) and (n, L_2) such that L_1 dominates L_2, i.e., $L_1 \subseteq L_2$.

5.2.2 Logic Rewriting

Logic rewriting is an efficient optimization method. A large part of its runtime can be reduced to cut enumeration, which scales well even to very large logic networks [98]. It proceeds roughly as follows. First, given a logic network, we partition it into smaller subnetworks, typically using a cut enumeration algorithm. We then have some source from which we draw, for any subnetwork N, a list L of potential replacement networks. The replacements correspond to alternative, possibly optimum, implementations of $F(N)$. If a replacement network $M \in L$ leads to a local improvement, we substitute it for N. Naturally, we try to pick the M such that leads to the best improvement, i.e. such that $\text{improvement}(N, M') \leq \text{improvement}(N, M)$, for all $M' \in L$.

Generally speaking, a rewriting algorithm may source the replacement networks in arbitrary ways.[1] In this chapter and the next we discuss several options based on exact synthesis. The first is to precompute a two-level index from functions to a list of their optimum Boolean chains, as described in Chapter 4. The second is to avoid precomputation, and to build such an index *on-demand*, storing data only for those functions that we encounter in practice. This avoids storing a large number of random functions that remain unused. Finally, we may use exact synthesis completely on-the-fly, by never storing any information in a database and computing replacement networks at runtime. This last option is discussed in Chapter 6.

The notion of *DAG-aware* logic rewriting was introduced in [93]. A DAG-aware rewriting algorithm takes into account the shared structure of the subject graph and replacement subnetworks. By exploiting this shared structure, it can find additional optimizations that may be missed by other, DAG-unaware, rewriting algorithms. For instance, suppose we are given two replacement networks M and K for some subnetwork N. Suppose further that $\sigma(M) > \sigma(K)$. We may then still wish to replace N by M rather than K, even though it would add more nodes. To see how this is possible, see Figure 5.1. It shows an example where, even though M uses more nodes than K to implement f, choosing M requires adding fewer nodes to the graph, by relying on shared logic.

Let us review how a DAG-aware rewriting algorithm might be implemented. The algorithm to compute the improvement, or *gain* makes use of reference counting and assigns a value to each node in the network. These values are initialized with the nodes' fanout sizes. New

[1] Assuming, of course, that they correctly realize $F(N)$.

nodes that are added to the network for a possible replacement will be assigned a reference count of 0. The reference count of a node indicates how many other nodes require this node in the network. In particular, a reference count of 0 means that the node is not required in the network. We may also exploit structural hashing [92], i.e., nodes from a replacement candidate that are already in the network will not be added another time, and also its reference counter will not be changed.

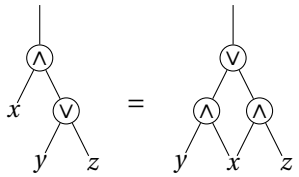
To simulate the removal of a node n from a network, we recursively decrement all predecessors in the transitive fanin of the node and continue as long as the reference counters of a child become 0 or a leaf node is reached. Algorithm 9 shows the details. It receives as inputs the node n and the leaves of a cut I.

Algorithm 9 Dereferencing a node

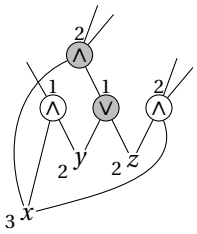
```
function DerefNode(n, I)
    if n ∈ I then
        return 0
    end if
    value ← 1
    for each child c of n do
        ref(c) ← ref(c) − 1
        if ref(c) = 0 then
            value ← value + DerefNode(c, I)
        end if
    end for
    return value
end function
```

Adding a node to a network can be simulated by the inverse algorithm to DerefNode, called RefNode (see Algorithm 10), which will increment reference counters and continue on the predecessors as long as the reference counter was 0 before incrementing it, and stops otherwise or when it reaches a leave node.

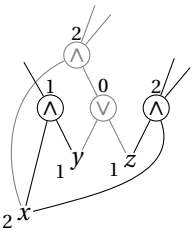
Figure 5.1 shows the gain calculation of substituting one potential cut subnetwork for another. Figure 5.1(a) shows two functionally equivalent subnetworks. The cut on the left is already contained in the network shown in Figure 5.1(b). Figure 5.1(b) also shows the initial reference counters which are equal to the fanout size of each node. Calling DerefNode on the top most AND gate changes the references counters as shown in Figure 5.1(c). In particular, the OR gate in the middle of the network now has a reference value of 0, meaning it is not required anymore after deleting the cut. Together with the root node this leads to a value of 2 which is returned by DerefNode. Afterwards the logic for the replacement cut is added in Figure 5.1(d). Note that two of the three gates are already present in the network and only one new node is added, which is initialized with a reference value of 0. All other reference values remain the



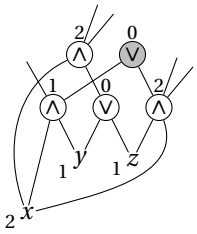
(a) Replacement candidate



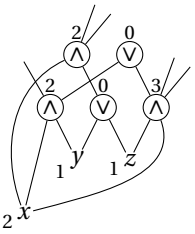
(b) Initial network



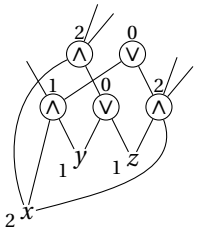
(c) Deref original cut, obtained value 2



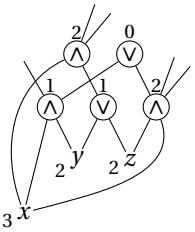
(d) Insert replacement candidate



(e) Ref replacement cut, obtained value is 1



(f) Deref replacement cut, obtained value is 1



(g) Ref original cut, obtained value is 2

Figure 5.1 – Estimating the gain of a replacement cut using reference counting.

Algorithm 10 Referencing a node

```
function RefNode(n, I)
    if n ∈ I then
        continue 0
    end if
    value ← 1
    for each child c of n do
        ref(c) ← ref(c) + 1
        if ref(c) = 1 then
            value ← value + RefNode(c, I)
        end if
    end for
    return value
end function
```

same. Calling RefNode on the root node of the inserted cut simulates an insertion of the cut and leads to the reference values as in Figure 5.1(e). The function returns 1 for the increment of the root node. From these two values we can derive that replacing the first cut by the other will save $2-1=1$ nodes. Since the cost of the replacement should only be calculated and not actually be performed one can undo the changes to the reference counters by simply calling the inverse functions in inverse order, i.e., calling DerefNode on the root node of the replacement cut and RefNode on the root node of the original cut leading to the reference values as shown in Figure 5.1(f) and Figure 5.1(g), respectively.

Algorithm 11 Adding a new cut (n', I) into the network and calculating the gain when replacing an existing cut (n, I).

```
function DryReplace(N, n ↦ n', I)
    v1 ← DerefNode(n, I)
    Insert cut (n', I) into the network
    v2 ← RefNode(n', I)
    DerefNode(n', I)
    RefNode(n, I)
    return v1 − v2
end function
```

This example motivates a function called DryReplace$(N, n \mapsto n', I)$ that inside a network N simulates the replacement of an existing cut (n, I) with a new cut (n', I) by using reference counters. The algorithm does not change the reference values of existing nodes in N and all newly added nodes will be assigned a reference value of 0. The function returns the gain of replacing the existing cut with the new one. This gain may be negative. We will see examples of how DryReplace can be used in Section 6.3.

Algorithm 12 Compute the size of the MFFC of n.

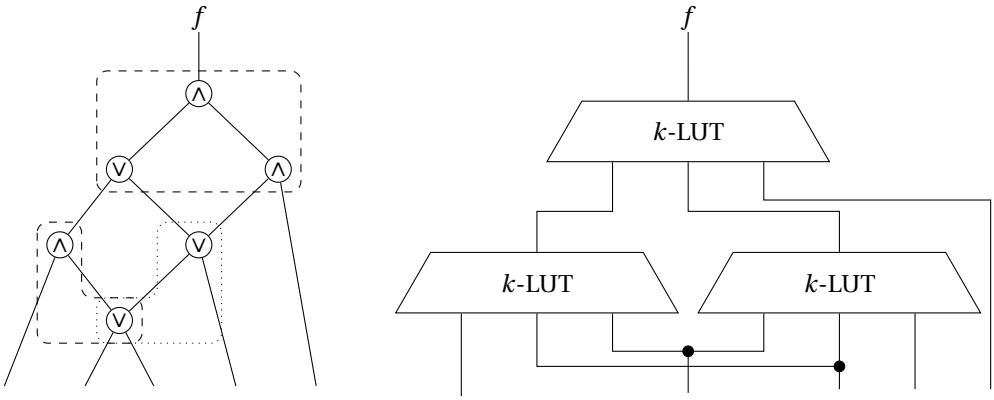
```
function MFFCSize(N, n)
    I ← primary inputs of N
    v ← DerefNode(n, I)
    RefNode(n, I)
    return v
end function
```

The routines RefNode and DerefNode can also be used conveniently to compute the size of the *maximum fanout-free cone* (MFFC) of a node, as shown in Algorithm 12. To achieve this, we let the cut leaves I be the primary inputs of the network, in order to find all logic in the node's MFFC.

Finally, rewriting a logic network may be achieved by the pseudocode presented in Algorithm 13.

Algorithm 13 A logic rewriting algorithm.

```
function LogicRewrite(Logic network N, cut size l, cut limit p)
    C ← CutEnumeration(N, l, p)
    for each gate n ∈ N in topological order do
        if MFFCSize(N, n) = 1 then
            continue
        end if
        bestGain ← 0
        bestReplacement ← Λ
        for each cut (n′, L) ∈ C(n) do
            gain ← DryReplace(N, n ↦ n′, L)
            if gain > bestGain then
                bestGain ← gain
                bestReplacement ← n′
            end if
        end for
        if bestReplacement ≠ Λ then
            Replace(N, n ↦ n′, L)
        end if
    end for
end function
```



(a) k-cuts in a Boolean network. (b) Corresponding k-LUT network.

Figure 5.2 – Any k-feasible cut can be implemented by a single k-LUT. In this example $k = 3$.

5.2.3 LUT Mapping

Technology mapping, also known as *cell-library binding* [88], is the problem of efficiently mapping a logic network to one in which nodes correspond to functions from some technology library. For example, we may want to map an MIG to a network of NAND2 gates, which is then converted into a CMOS circuit during physical synthesis. LUT mapping is a special case of technology mapping, in which map the subject graph to a network of k-bounded lookup tables (k-LUTs). A k-LUT is a lookup table with k inputs. It is a powerful logic primitive which can represent any function on k variables. The k-LUT model is commonly used to map logic networks to FPGAs.

There is a close correspondence between the subnetworks created by cut enumeration and k-LUTs: since a k-LUT can be used to represent any k-feasibly cut function, we can use a single k-LUT to represent each subnetwork. Figure 5.2 visualizes this correspondence. Therefore, the problem of k-LUT mapping can be reduced to finding efficient k-LUT covers of a logic network. In other words, after we perform cut enumeration, k-LUT mapping is the task of selecting a subset of k-cuts that cover the entire network. Such a set then represents a k-LUT network that is functionally equivalent to the original logic network. During cut selection we may want to optimize different objectives. For instance, to perform a delay-oriented k-LUT mapping, we want to select those cuts that minimize the critical path. An area-oriented mapping, on the other hand, tries to minimize the number of selected cuts.

A breakthrough in delay-oriented LUT mapping occurred when Cong and Ding introduced the FlowMap algorithm [34]. It was the first algorithm to show how k-feasible cuts can be used to obtain a minimum-depth k-LUT cover. Several improvements of FlowMap have since been made. Some of these improvements include generalizing the algorithm to a

more general cut enumeration basis, improving the runtime and memory requirements, as well as improving different aspects of the final cover such as area reduction [98, 35, 36, 37, 30]. Although depth-optimal LUT mapping is a solved problem, area-optimal LUT mapping is NP-hard and remains an open problem [48]. However, different effective area-recovery heuristics have been proposed [30, 84]. More recently, technology mapping algorithms have aimed at reducing the problem of structural bias [28, 29]. This problem refers to the, possibly suboptimal, bias imposed by cut-based mapping algorithms. Cuts are derived from a specific network structure. Hence, if the network structure is a suboptimal decomposition of the logic network function, the resulting LUT (or standard cell) cover may be suboptimal as well.

5.3 Contributions

As we discuss in Section 5.2.2, some logic rewriting algorithms use exact synthesis to replace small subnetworks by their optimum representations. However, conventional approaches can suffer from two major drawbacks. First, their scalability may be limited, as they may enumerate all (or a large set of) Boolean functions to precompute their optimum representations. However, the space of Boolean functions is so large that enumeration quickly becomes intractable. Moreover, we would prefer not having to store a large number of replacement networks. Therefore, in practice such rewriting methods are limited to functions on 4 variables, or have to give up exactness for heuristic results [93, 81, 147]. Second, the strategies used to decide which subnetworks to rewrite are not necessarily optimal. In other words, the heuristics by which the algorithms choose which subnetworks to substitute may miss some opportunities.

In this chapter, we propose a novel method which aims to mitigate some of the difficult problems facing by rewriting algorithms. It does so by using of the following techniques:

1. *We propose an alternative subnetwork selection strategy based on LUT mapping.* In doing so, we find that, for some networks, using LUT mapping heuristics can be advantageous as compared to standard selection strategies.

2. *We show how enumeration of the Boolean space can be avoided, allowing our method to scale to larger subnetworks.* We achieve this by computing optimum representations only for functions that occur in practice. We find these functions by using LUT mapping and NPN canonization as filters. This also allows us to construct a database of (classes of) Boolean functions that occur in practice. Further, the filters select exactly those functions that are to be included final optimized network. Therefore, we avoid computing optima for functions that will not be used. We refer to this process of collecting “practical” functions as *mining* for Boolean functions. An added benefit of our approach is that, for some networks, LUT covers turn out to select better covers than those found by previous

approaches, thus improving the subnetwork selection strategy.

3. *We introduce the compact XMG data structure as a novel logic representation.* Furthermore, we use the XOR and MAJ operators as the primitive set $\mathscr{B}$ in our exact synthesis algorithm. Since XMGs enable compact logic representation, using these primitives in exact synthesis reduces its runtime as compared to AIGs or MIGs, especially when combined with improvements to the exact synthesis algorithm introduced in [129].

Using our method, we show a **45.8%** geometric mean reduction (taken over size, depth, and switching activity), a **6.5%** size reduction, and depth · size reductions of **8.6%**, as compared to academic state-of-the-art algorithms. Finally, we outperform 3 over 9 of the best known size results for the EPFL benchmark suite, reducing size by up to **11.5%** and depth up to **46.7%**.

5.4 XOR-Majority Graphs

The central Boolean operators in this chapter are the exclusive OR (XOR) and the ternary majority (MAJ). Recall from Section 1.1 that we define the XOR operator as

$$x \oplus y = x\bar{y} \vee \bar{x}y = (x \vee y)(\bar{x} \vee \bar{y})$$

and the MAJ operator as:

$$\langle xyz \rangle = xy \vee xz \vee yz = (x \vee y)(x \vee z)(y \vee z).$$

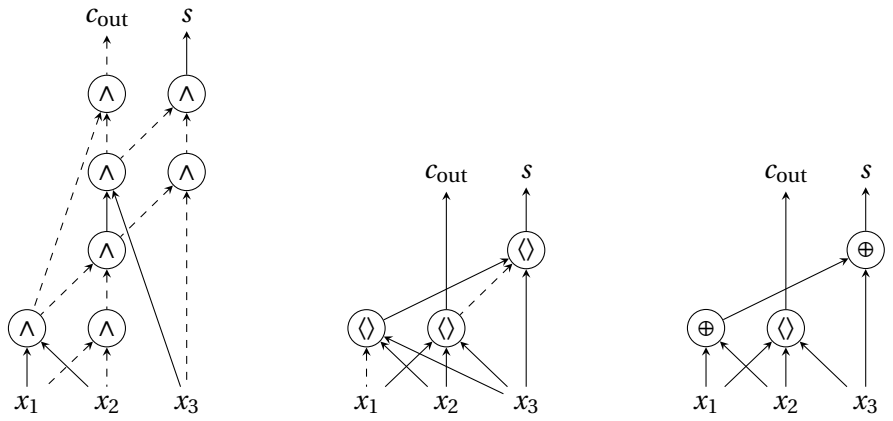
The MAJ operator has many interesting properties [20, 33, 66]. Also known as the *median* operator, Knuth refers to it as the "probably the most important ternary operation in the universe" [69, p. 63]. Notably, MAJ is self-dual [2]. We say that a function f is self-dual if

$$\overline{f(x_1, x_2, \ldots, x_n)} = f(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n).$$

In other words, f is self-dual if and only if we invert its output by inverting the polarities of its inputs. The related property known as *inverter propagation*, which is implied by self-duality, means inverters can be propagated through networks of majority nodes. Interestingly, there is a similar relationship between MAJ and XOR operators. Namely, XORs propagate through MAJ operators, much like inverters do:

$$a \oplus \langle xyz \rangle = \langle (x \oplus a)(y \oplus a)(z \oplus a) \rangle$$

The XOR operation inverts one of its operands if the other one is set to 1, i.e., $x \oplus 1 = \bar{x}$. Hence,



(a) AIG (b) MIG (c) XMG

Figure 5.3 – Size-optimum full adders, given in AIG, MIG, and XMG representations, respectively. Dashed lines indicate complemented edges. We see that $\sigma(a) \leq \sigma(b) \leq \sigma(c)$.

although XOR is not self-dual, it does allow us to propagate inverters, since:

$$x \oplus y = \bar{x} \oplus \bar{y} = \overline{\bar{x} \oplus y} = \overline{x \oplus \bar{y}}$$

and:

$$\bar{x} \oplus y = x \oplus \bar{y} = \overline{x \oplus y} = \overline{\bar{x} \oplus \bar{y}}.$$

This interesting relationships between MAJ, XOR, and inverters have inspired the development of XMGs. Originally introduced in [53], XMGs are logic networks in which each gate corresponds to either a MAJ or a XOR operator. To represent inversion, we also allow complemented edges in the graph. Hence, XMGs are extensions of the MIGs introduced in [7]. In XMGs both inverters and XOR nodes can propagate freely through the MAJ nodes. They are more expressive, and therefore more compact, than AIGs or MIGs. This makes them well suited for use in an optimization flow based on exact synthesis: compact representations allow for smaller solutions, which can be found more quickly, as we have seen in Chapter 2.

Figure 5.3 shows an XMG representation for a full adder, next to equivalent optimum-size AIG and MIG representations. MAJ and XOR nodes are represented by nodes with 3 and 2 outgoing edges, respectively. We can see that the XMG representation requires fewer nodes. This will be the case in general as well. To see why, consider an arbitrary function $f : \mathbb{B}^n \to \mathbb{B}^m$. Let X, A, and M be its optimum XMG, AIG, and MIG representations, respectively. Then we clearly have $\sigma(X) \leq \sigma(M) \leq \sigma(A)$, since AIGs are included by MIGs, which in turn are included by XMGs.

5.5 Optimization Method Overview

Figure 5.4 gives an overview of our proposed method. In this chapter, our primary goal is size optimization, but the method could easily be adapted to target other objectives. It is applicable to any k-bounded network, i.e., a network in which each gate has at most k inputs. Note that, if a network is not k-bounded, it may be decomposed to obtain a functionally equivalent k-bounded network [34]. Therefore, in the sequel, we will assume, without loss of generality, that input networks are k-bounded. In Section 5.6, we describe in detail our specialization of this method for XMG size optimization.

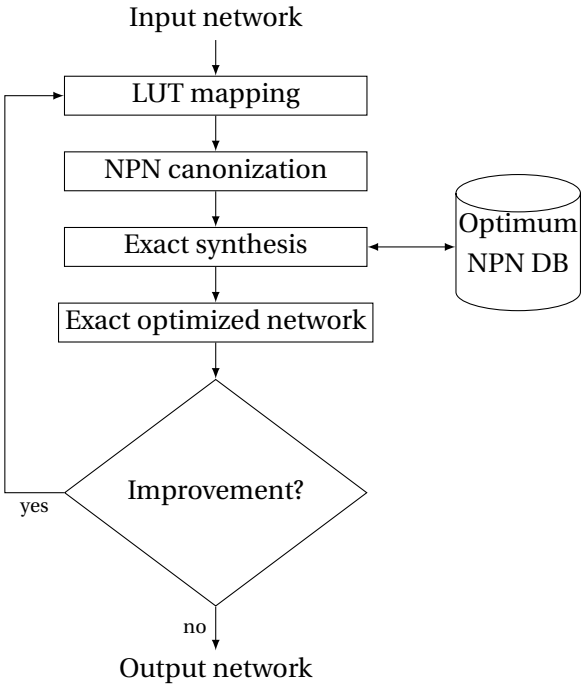


Figure 5.4 – An overview of the optimization flow.

The input to our method is a parameter k and a k-bounded logic network N. We first perform LUT mapping on N in order to find a suitable k-LUT cover. As our goal is size optimization, a suitable cover is one that minimizes the number of LUTs, and we use the appropriate heuristics to obtain it. After finding a cover, we compute the NPN classes for the functions of the LUTs in the cover. We then invoke exact synthesis for these NPN classes, producing locally optimum subnetworks. The results of exact synthesis are saved in a database that stores the optimum representations of the NPN classes we have encountered. These results may be reused in subsequent iterations. Note that, in essence, this database fulfills the role of object S from Section 4.3.2. However, instead of precomputing the object, it is constructed on-demand, as

new NPN classes are encountered. Finally, the locally optimum networks are merged together to create an optimized, functionally equivalent, network N'. This optimization process may be iterated on N' to improve results. Applying this method with larger k increases the size of the subnetworks that we optimize. Larger k enable better optimization results, on average. To see why, note that in the extreme case k is equal to the number of primary inputs of the network. The result would then be the optimum representation of that network. Hence, we would like to apply this method to the largest possible values for k.

The reason for using NPN canonization and the on-demand construction of S is twofold. First, it saves storage space, since we only encounter a small fraction of the total number of 2^{2^k} functions in practice. The number of practically occurring NPN classes is smaller still. Second, it saves computation time. We only invoke exact synthesis on those NPN classes we encounter, not for the entire space $\mathbb{B}^{2^k}$. Moreover, exact synthesis is invoked only once per class. In all subsequent optimization iterations, the result can simply be retrieved from the database, which requires only a negligible amount computation time as compared to the rest of the optimization flow.

5.5.1 Comparison to Previous Work

Our optimization method has some similarities to earlier AIG rewriting optimizations [93, 81, 147]. These methods also find k-feasible cuts to obtain replacement subnetworks. One difference is that our method does not rely on the enumeration of Boolean functions and their optimum subnetworks. This is one of the key differences which allows our method to scale. Enumeration of functions becomes impractical for $k > 4$. For example, there are 2^{2^6} 6-variable functions, with 200,253,952,527,185 corresponding NPN classes. Suppose that the average computation time required to find the optimum representation for these functions is 0.002 seconds.[2] Even if we were to obtain, through some oracle, a list of the NPN classes, it would still take over 12,700 years to synthesize all their optimum representations. Therefore, avoiding enumeration is crucial to obtain tractable runtimes. We avoid it by computing optima only for those NPN classes that occur in a cover. Thus, we only examine a small portion of the total number of Boolean functions. In other words, we *mine* the space of "useful" Boolean functions that occur in practice. This greatly reduces the computation time required by our approach, and makes exact synthesis tractable for $k > 4$. For example, when mining the EPFL benchmarks for 6-variable functions we only find 286 unique NPN classes. Thus, we reduce the number of exact synthesis invocations by twelve orders of magnitude.

Another difference between our method and previous approaches is in the subnetwork selection heuristic. We select those subnetworks that appear in the LUT cover. This turns out to

[2] This number is based on experiments determining the runtime of our algorithm on 4-variable functions. On 6-variable functions the average runtime would be higher.

be a selection heuristic that, for some networks, compares favorably to the heuristics used by [93, 81]. Those approaches rely on purely local information, whereas LUT mapping may use heuristics with a global view. Thus, LUT mapping improves subnetwork selection.

The approach in [147] also mines for useful circuit structures. However, it is aimed at depth optimization, whereas we focus on size. Additionally, the results in [147] are not necessarily exact, but are rather based on mining the results of heuristic optimizations.

Finally, our approach is distinct from *remapping* methods [17, 114]. For example, the method in [17] iteratively improves *mapped* circuits, by symbolically optimizing Boolean relations with a specified cell library. In contrast, ours is a technology independent logic optimization method that uses SAT or SMT for optimization. Additionally, the runtime of our method may be improved by mining circuits for useful functions in advance.

5.6 Method Implementation

The optimization method described in Section 5.5 is a generic one. It could be applied to arbitrary logic networks and tuned to support different optimization objectives. In this section, we describe a particular instantiation of this method, in which we specialize it to XMG size optimization and use an SMT solver for exact synthesis.

5.6.1 Exact Synthesis

Here, we describe a CNF encoding for the synthesis of XMGs. Given the detailed descriptions of encodings in Chapter 2, this encoding should not contain too many surprises, but there are still some interesting differences. We describe them here.

Recall that the objective of our encoding is, given a Boolean function $f(x_1, \ldots, x_n)$, to decide if there exists an XMG of size r. Our XMG encoding $\mathscr{F}_r$ consists of the following variables, for $n < i \leq n + r$ and $0 \leq t < 2^n$:

$$
\begin{aligned}
x_{it} &: \text{t}^{\text{th}} \text{ bit of } x_i\text{'s truth table} \\
x_{it}^{(k)} &: \text{t}^{\text{th}} \text{ bit of } x_i\text{'s input truth table for } 1 \leq k \leq 3 \\
s_{1j}, s_{2k}, s_{2l} &: x_i = \circ_i(x_j, x_k, x_l) \text{ for } 0 \leq j, k, l < i \\
g_i \in \mathbb{B} &: \circ_i \text{ is MAJ if } g_i = 1 \text{ and XOR otherwise} \\
p_{ki} \in \mathbb{B} &: \text{ gate } i \text{ input } k \text{ is complemented iff } p_{ki} = 0 \text{ for } 1 \leq k \leq 3 \\
p \in \mathbb{B} &: \text{output polarity}
\end{aligned}
$$

In this encoding, the s variables again correspond to selection variables, and decide for each gate i what its inputs are. An important difference between this encoding and those presented in Section 2.2 are the p_{ki} variables. They can be viewed as an augmentation to the selection variables. We say that the k-th input of step i is complemented if and only if $pki = 0$. Thus, these variables capture the XMG's complemented edges. Another new aspect to this encoding are the *gate type* variables. In an XMG, gates may correspond to either XOR or MAJ operators. We encode this using the g_i variables. Truth table simulation is achieved in a manner similar to the DITT encoding described in Section 2.2.3. The variables $x_{it}^{(k)}$ correspond to the truth tables of the inputs to step i. The truth table computed by step i is captured by variables x_{it}. It depends on the input truth tables, the operator implemented by i, and the input polarities. In this chapter, we rewrite only single-output subnetworks (i.e. cuts). Therefore, our encoding only needs to consider single-output Boolean functions. Hence, we can assume that the output is $f = x_{n+r}$, such that it points to the last gate in the XMG. Consequently, we need add one additional variable p to the encoding. This variable represents the output polarity.

To ensure correct functionality of synthesized XMGs we add the following clauses. The constraint:

$$x_{it} \equiv \left(g_i \,?\, \langle x_{it}^{(1)} x_{it}^{(2)} x_{it}^{(3)} \rangle : (x_{it}^{(1)} \oplus x_{it}^{(2)})\right) \tag{5.1}$$

ensures that each gate computes either MAJ or XOR depending on the value of g_i. The constraints:

$$(s_{ki} = j) \rightarrow (x_{it}^{(k)} \equiv x_{jt} \oplus \bar{p}_{ki}) \qquad \text{for } 1 \leq k \leq 3 \tag{5.2}$$

selects the input truth bits for the k-th input of step i. Depending on the value of the polarity variable p_{ki}, the inputs may be complemented. Thus, Equation (5.1) and Equation (5.2) ensure that values are propagated correctly through the XMG which is represented by the s- and p-variables. Note that, in Equation (5.2), j ranges from 0 to $i-1$. Finally, we define $b_0^{(t)} = 0$ for all t, such that step 0 represents the zero constant. The constraint:

$$x_{(n+r)t} \equiv (f(t) \oplus \bar{p}) \tag{5.3}$$

ensures that the last gate computes the correct function value at truth table row t.

> ***Example.*** Suppose we want to check if there exists an XMG with $r = 2$ gates for the function $f = x_1 \,?\, x_2 : x_3$. In other words, we want to synthesize an XMG for the 3-input *if-then-else* function which selects either x_2 or x_3, depending on the value of x_1. This function is also commonly known as a 2-to-1 multiplexer. Thus, $n = 3$ in this case. The SAT instance contains 6 variables s_{14}, s_{24}, s_{34}, s_{15}, s_{25}, s_{35}, 6 variables p_{14}, p_{24}, p_{34}, p_{15}, p_{25}, p_{35}, 2 variables g_4, g_5, and the variable

p. It contains 48 $x_{it}^{(k)}$-variables for the input truth tables and 16 x_{it}-variables, as t ranges from 0 to 7. There are 16 constraints of type (5.1), 216 constraints of type (5.2), and 8 constraints of type (5.3). The SAT instance is satisfiable and as satisfying assignments we may obtain

$$\begin{aligned}
&g_4 = 0, g_5 = 1\\
&s_{14} = 1, s_{24} = 3, s_{34} = *\\
&s_{15} = 2, s_{25} = 3, s_{35} = 4\\
&p_{14} = 1, p_{24} = 1, p_{34} = 1\\
&p_{15} = 1, p_{25} = 1, p_{35} = 1\\
&p = 1
\end{aligned}$$

which corresponds to the XMG

$$x_4 = x_1 \oplus x_3 \qquad x_5 = \langle x_2 x_3 x_4 \rangle$$

with $f = x_5$. Hence, by synthesizing this function, we have found the interesting identity

$$x_1 \,?\, x_2 : x_3 = \langle (x_1 \oplus x_3) x_2 x_3 \rangle.$$

Note that s_{34} may be assigned any value in between 0 and 3 by the SAT solver since its value does not have an effect to the overall result as $g_4 = 0$.

The variables and constraints described so far suffice to make the algorithm work. We may add additional constraints to break symmetries. These may be based on properties specific to the operators we wish to synthesize. Using the commutativity and inverter properties of the majority operation can significantly reduce solving time. For example, in the case of a MAJ gate we can enforce that at most one of its operands is complemented. To see why, note that if a MAJ gate has two complemented inputs, we know from the self-duality property that such a gate is equivalent to a MAJ gate with a complemented output and one complemented input. Similarly, we can enforce that XOR gates do not use complemented operands. To achieve this, we can add the constraint:

$$g_i \,?\, \langle p_{1i} p_{2i} p_{3i} \rangle : p_{1i} p_{2i}.$$

We can also adopt other symmetry breaking constraints, such as those described in Section 2.3. For instance, we can ensure that no node can occur twice and that the operands of gates x_i and x_{i+1} are in co-lexicographic order for $n \leq i < n + r$.

5.6.2 XMG Size Optimization

Broadly speaking, given an input XMG N, our size optimization algorithm consists of the following stages:

1. Area-oriented k-LUT mapping of N
2. NPN canonization of the functions in the k-LUT cover
3. Decomposing the k-LUTs into locally optimum XMGs
4. Merging the locally optimum XMGs into an optimized XMG N'

These steps are iterated until N' no longer improves.

In the first step of our algorithm, we use our LUT mapper to generate an area-oriented cover. We use the area-flow and exact-area heuristics [84]. The reason for creating a LUT cover is that it turns out to be a superior subnetwork selection strategy as compared to previous approaches. Area-oriented selection using area-flow and exact-area selects a minimal number of LUTs to cover the entire network, using both a global and local view of the network. Thus, LUT mapping is a subnetwork selection strategy that takes both local and global information into account. It is also a good starting point for size minimization. The fewer LUTs (cuts) we need to decompose, the fewer nodes the resulting optimized XMG will have. Finally, by mapping into a minimal number of LUTs, we minimize the number of functions on which we have to invoke our exact synthesis algorithm.

After generating a cover, we extract an optimized XMG. We do so through a topological traversal of the nodes selected in the cover. We compute the NPN canonization of the cut functions, and obtain its optimum XMG. If the optimum XMG is not already present in the database, we compute it and store the results in the NPN class database. We use the distributed key-value database Redis [116] to store optimum XMGs. The pseudocode for this procedure can be found in Algorithm 14.

5.7 Experimental Evaluation

We have integrated the proposed algorithm into our C++ logic synthesis frameworks. The experiments have been carried out Intel E5-2680 CPU with 2.50 GHz with 64 GB of main memory running Linux 3.13.

Algorithm 14 An XMG size optimization procedure using LUT mapping and exact synthesis. Receives as input logic network N and LUT size k. Returns optimized logic network N'.

```
function XMGOPTIMIZE(N, k)
    N' ← N
    repeat
        N ← N'
        N' ← new_xmg()
        Perform area-oriented mapping of N into k-LUTs
        for each primary input i in N do
            create_input(N', i)
        end for
        for each LUT l in the cover in topological order do
            f ← function computed by l
            npn ← NPN_canonization(f)
            opt_xmg ← database_get(npn)
            if opt_xmg = nil then
                opt_xmg ← exact_xmg(npn)
                database_save(npn, opt_xmg)
            end if
            create_node(N', n, opt_xmg)
        end for
    until size(N') ≥ size(N)
    return N'
end function
```

5.7.1 XMG Size Optimization

In this experiment, we compare XMG size optimization to AIG size optimization. Our implementation reads the description of a combinational circuit, reduces the size of the circuit by using the techniques described in Section 5.6.2, and writes back an optimized circuit. We compare our results to those obtained by the state-of-the-art academic logic synthesis package ABC 1.01 [23]. Using ABC, we iteratively apply its *resyn2* script until results no longer improve. We measure the size, depth, and switching activity of the resulting optimized networks. The benchmarks are taken from the EPFL benchmark suite, which contains combinational circuits in AIGER format. All results have been formally verified with ABC's *cec* command. Table 5.1 shows the results.

We show the results for our procedure with $k = 4$, $k = 5$, and $k = 6$. On average, the {size, depth, activity} of XMGs is smaller by {21.7%, 32.1%, 6.1%}, {22.6%, 33.9%, 3.8%}, and {39.4%, 42.2%, 27.7%} for $k = 4$, $k = 5$, and $k = 6$, respectively. Using a size·depth·activity figure of merit, XMG optimization performs 50.1%, 50.8%, and 75.3% better than AIGs for $k = 4$, $k = 5$, and $k = 6$, respectively. We also compute the geometric mean, taken over the sizes, depths,

Table 5.1 – Comparing XMG and AIG size optimization.

Size Optimization		**XMG** ($k = 4$)			**XMG** ($k = 5$)			**XMG** ($k = 6$)			**AIG**		
Benchmark	I/O	Size	Depth	Activity	Size	Depth	Activity	Size	Depth	Activity	Size	Depth	Activity
Adder	256/129	639	130	888.5	575	131	794.8	383	129	508.9	1019	255	981.7
Barrel shifter	135/128	3281	16	3269.5	2932	18	3294.2	2858	17	3625.4	3141	12	2546.4
Divisor	128/128	29607	4371	21649.5	29607	4371	21649.5	39768	4310	31997.9	40698	4361	31431.1
Hypotenuse	256/128	155349	12507	157711.6	143282	12845	153816.8	99927	9017	100709.1	211262	24670	169609.8
Log2	32/32	27936	275	21862.4	30574	267	26228.5	23006	219	17626.5	29238	375	19046.5
Max	512/130	2296	296	2593.9	2183	258	2569.9	1982	254	2442.3	2831	151	2630.3
Multiplier	128/128	17508	154	17300.7	18771	150	19702.0	16575	136	16675.0	24554	262	20169.1
Sine	24/25	5100	176	3956.0	5419	225	3521.6	3825	121	2620.5	5010	160	3127.6
Square-root	128/64	20130	6031	19456.0	24570	5058	21184.1	17369	6149	17441.92	19437	4968	16977.8
Square	64/128	15070	130	13632.3	15724	132	16021.9	8527	155	8335.0	16568	247	12779.5
Average:		27691.6	2408.6	26232.0	27363.7	2345.5	26878.3	**21422**	**2050.7**	**20198.3**	35375.8	3546.1	27929.9
Geom. Mean:		3555.5			3610.5			**3009.0**			3736.8		

and switching activity of the networks. Both our method and ABC start with the same input networks, containing only AND gates. However, by doing exact synthesis, our method is able to more effectively compress subnetworks, due to the expressive logic primitives in the XMG representation. In other words, our algorithm effectively takes advantage of XMG expressivity. Furthermore, these results confirm our intuition that synthesizing larger subnetworks leads to a better result overall. Higher k lead to better results in logic optimization. Finally, one might object that the more expressive XMG primitives are bound to result in smaller representations, thus making these results unsurprising. However, as the next experiments show, the XMG size optimization advantage also carries over into LUT mapping improvements.

5.7.2 LUT Mapping

Our previous experiment compares XMGs to AIGs in a logic optimization context. In order to further investigate the potential of our size optimization method, we evaluate the results after k-LUT technology mapping. We compare the results of 6-LUT mapping of the optimized networks from Table 5.1. As the networks are optimized for size, we focus on area-oriented technology mapping. All networks were mapped with ABC, using the command *if -a -K 6*.

Table 5.2 summarizes the results of k-LUT technology mapping. Compared to the AIG flow, XMG flow reduces mapped network size by 6.5%, 3.8%, and 9.4% for $k = 4$, $k = 5$, and $k = 6$, respectively.

Table 5.2 also shows two other figures of merit. First, the geometric mean, taken over the sizes and depths of the mapped networks. The geometric means of the XMG mapped networks are lower by 45.8% for $k = 4$, 36.8% lower for $k = 5$, and 48.9% lower for $k = 6$, as compared to the mapped AIGs. Second, it shows the size$\cdot$depth measure. With this measure, XMGs optimized with $k = 4$ show an 8.6% improvement as compared to AIGs, while $k = 6$ gives a 8.1% improvement. This is caused by the fact that the mapping heuristics work out such that $k = 4$ leads to smaller depth. As depth is not our main objective here, we do not consider this

Table 5.2 – Comparing 6-LUT Mapping for XMGs and AIGs

	XMG ($k = 4$)		**XMG** ($k = 5$)		**XMG** ($k = 6$)		**AIG**	
Bench	Size	Depth	Size	Depth	Size	Depth	Size	Depth
Adder	251	131	192	64	250	122	249	121
Bar	888	6	532	5	512	4	512	4
Div	12094	2123	12094	2123	12640	2087	8190	2058
Hyp	50835	7964	52376	8506	48772	8401	47508	8339
Log2	8438	162	8965	152	7961	157	7721	152
Max	745	118	741	121	710	122	771	66
Mult	5700	127	5498	127	5685	126	5689	126
Sine	1655	78	1450	71	1487	76	5615	73
Sqrt	6595	2144	8084	3957	6366	2237	5130	2211
Square	3969	122	3839	121	3930	120	16057	122
Average:	9117.0	**1296.7**	9377.1	1524.7	**8831.3**	1345.2	9744.2	1327.2
Geomean:	904		1055		**851**		1669	
Size · depth:	**11822013.9**		14297264.4		11879864.8		12932502.2	

to be an issue.

5.7.3 Comparison To Best Known Results

The previous experiments show that XMGs compare favorably to AIGs in an optimization and synthesis flow for k-LUTs. We now turn to a comparison with the *best known* results for these benchmarks [3]. Published alongside the benchmarks of the EPFL benchmark suite, are two sets of *best known results*. These are the best known 6-LUT covers (for both size and depth) for the benchmarks in the suite. These results may be obtained by *any* method. As such, the techniques used to obtain these results were not limited one method, but consist of a combination of advanced ABC scripts. In fact, we do not know for each benchmark exactly which method was used to obtain its best known 6-LUT cover. However, all covers have been formally verified. Hence, they serve as a good point of reference.

In this experiment we again map our optimized XMGs to 6-LUTs using ABC. However, we are now comparing against the best known results for area-oriented 6-LUT mapping. As these have been obtained in various ways, we do not limit ourselves to one type of mapping. We use ABC's *if* command to obtain both area-oriented and depth-optimal results. We then collect the best mappings from these and compare them to the best known results. The results can be seen in Table 5.3.

[3] As of July 15th 2016

Table 5.3 – Comparing Best XMGs To Best Known 6-LUT Mapping Results.

	Best Known Results		**XMG Mappings**	
Benchmark	Size	Depth	Size	Depth
Adder	201	73	**192**	**64**
Barrel shifter	512	4	512	4
Divisor	**3813**	1542	10670	**864**
Log2	**7344**	142	7893	**87**
Max	**532**	192	846	**72**
Multiplier	5681	120	**5245**	**64**
Sine	**1347**	62	1488	**48**
Square-root	**3286**	1180	5014	**1032**
Square	3800	116	**3364**	**85**
Average:	**2946.2**	381.2	3914.8	**257.8**
Geomean:	480		**437**	
Size · depth:	1123157.9		**1009151.9**	

We show significant improvements on three benchmarks, reducing the {size, area} of the Adder, Multiplier, and Square by {4.5%, 12.4%}, {7.7%, 46.7%}, and {11.5%, 26.8%}, respectively. For most other benchmarks, we are quite close in size, while substantially reducing depth. The main outlier to this trend is the Divisor benchmark. It is not obvious why this benchmark performs so poorly. One interesting observation is that our algorithm appears to work especially well on networks that do addition and multiplication. Networks such as Divisor and Square-root correspond to the inverse of these operations, and our algorithm performs less well on these.

We again calculate the geometric means over the sizes and depths. Our results improve on the mean by 9% as compared to the best results. Using the size·depth measure, we show a 10.1% improvement.

5.8 Summary

In this chapter, we have investigated the application of exact synthesis to the optimization of large Boolean functions. We have described an optimization method based on a combination of LUT mapping, NPN canonization, and exact synthesis. Our method also introduces Boolean function mining, a process which reveals what Boolean functions occur in practice, thus eliminating the need to (pre)compute and store all exact solutions. Finally, we have introduced XOR-Majority Graphs: a logic representation that enables compact logic networks, and hence faster exact synthesis.

The improvements described in this chapter have allowed us to design a novel algorithm which is a step in the direction of our goal of scaling up the use of exact synthesis in logic optimization. Our size optimization algorithm unlocks a 48.9% reduction in the geometric mean, a 9.4% average reduction in size, and a 8.6% reduction in LUT depth·size, as compared to the state-of-the-art ABC academic tool. It also outperforms 3 over 9 of the best known results for the EPFL benchmark suite, showing reductions of up to 11.5% in size and 46.7% in depth.

The method we propose here hints at a new potential research direction in which parallel and distributed computing power is invested to re-synthesize networks using exact synthesis methods. Distributed systems and compute clusters can be used to search and mine for optimum network representations. Such an environment also naturally lends itself to distributed or parallel exact synthesis using DAG topologies, as described in Chapter 3.

5.8.1 Future Work

The method described in this chapter is mostly concerned with size optimization. However, there are a number of possible extensions to this work that are worth considering:

- *Rewriting for depth/delay.* This is perhaps the most obvious extension. With some small modifications, our algorithm could be adapted for depth/delay rewriting.
- *Exact synthesis as a service.* Exact synthesis results can be shared and computed in the cloud. Although there exist a lot of k-input Boolean functions, we expect that only a small fraction of them occur in practice. Different users can access the same database to query for optimum networks. If the network has already been computed it can be returned immediately, otherwise, it is scheduled for computation.
- *Exact synthesis aware mapping.* The LUT mapping step in the optimization flow (Figure 5.4) decides for which subnetworks optimum representations need to be computed. If only a single of the optimum networks for these functions requires a large amount of runtime, exact synthesis becomes a bottleneck of the optimization. In such cases we could stop the computation and retry with another subnetwork selection strategy. For example, ranking functions by synthesis difficulty could be used to skip those functions that might be a bottleneck.

6 Optimizing Boolean Networks

In the previous chapter, we have taken a first look at optimizing large logic networks with exact synthesis. In this chapter, we present a more general optimization technique which also uses exact synthesis as one of the essential steps. In fact, it is a generalization of the DAG-aware rewriting algorithms that are also discussed in Chapter 5. The method we introduce here is applicable to all k-feasible Boolean networks, i.e. those networks whose nodes are k-input lookup tables (k-LUTs). AIGs can be viewed a special case of 2-feasible Boolean networks in which each node corresponds to an AND operator (with possibly complemented operands). This new method is a high-effort DAG-aware rewriting algorithm, called *cut rewriting*, which uses exact synthesis to compute replacements on the fly, with support for Boolean don't cares. Cut rewriting precomputes a large number of possible replacement candidates, but instead of eagerly rewriting the Boolean network, it stores the replacements in a conflict graph. Heuristic optimization is used to derive a best, maximal subset of replacements that can be simultaneously applied to the Boolean network from the conflict graph. Besides optimizing Boolean networks, our method can also be used to re-synthesize circuits that have already been mapped, even if these were initially optimized using some other method. To demonstrate this, we optimize LUT mapped Boolean networks obtained from the ISCAS and EPFL combinational benchmark suites. For 3-LUT networks, experiments show that we achieve an average size improvement of **5.58%** and up to **40.19%** *after* state-of-the-art Boolean rewriting techniques were applied until saturation. Similarly, for 4-LUT networks, we obtain an average improvement of **4.04%** and up to **12.60%**.

6.1 Introduction

Boolean rewriting is an optimization technique for large multi-level logic networks. More specifically, it is a type of logic rewriting algorithm, such as those described in Section 5.2.2. In Boolean rewriting, just as in many other logic rewriting algorithms, we iteratively select small

parts of the Boolean network and replace them with more compact representations. This reduces the overall number of nodes, while maintaining the global output functions of the Boolean network. DAG-aware AIG rewriting [93] is an efficient and well-known instantiation of logic rewriting, which is also discussed in Chapter 5. It exploits structural hashing to find beneficial replacements that utilize the existing logic within the network. Being DAG-aware allows one to obtain a gain even when replacing a smaller part of logic by a larger one, by reusing already existing logic in the network. AIG rewriting scales well, by combining efficient cut enumeration with fast truth table computations, and a database of precomputed replacement subnetworks.

Our goal in this chapter is to generalize DAG-aware AIG rewriting. To that end, we describe a DAG-aware rewriting algorithm that can be applied to k-feasible Boolean networks instead of AIGs. Replacements are computed on-demand using exact synthesis. This offers a more flexible, general, and scalable solution, as compared to methods that use a precomputed database, similar to the XMG-based optimization method described in Chapter 5. Recent achievements in SAT-based exact synthesis, such as the parallel topology-based synthesis described in Chapter 3 enable its integration as an efficient engine in various logic synthesis applications. As a consequence, the proposed approach is generic and capable of optimizing all common technology-independent logic representation including AIGs, MIGs, and XOR-based representations, as well as allows one to obtain size optimizations after technology mapping, e.g., in LUT mapping for FPGAs. Moreover, on the fly synthesis allows us to support don't care conditions, for which precomputing a database is intractable.

6.2 Preliminaries

As in Chapter 5, the algorithm and results we present in this chapter depend on prior notions such as cut enumeration, LUT mapping, and (DAG-aware) logic rewriting. Indeed, the preliminaries for the work in this chapter are essentially the same as those for Chapter 5. Therefore, for detailed descriptions of the above concepts we refer the reader to Section 5.2. We do briefly present some definitions and notation regarding Boolean networks here.

A *Boolean network* N is a *directed acyclic graph* (DAG). Each node corresponds to a logic gate. Each directed edge (n, m) is a wire connecting node n with node m. The *fanin*, respectively *fanout*, of a node $n \in N$ are the incoming, respectively outgoing, edges of the node. A Boolean network is *k-feasible* if the fanin size of all nodes is bounded by k. A k-LUT network is the most general k-feasible network in which each gate can implement an arbitrary Boolean function. The *primary inputs* (PIs) are the nodes of the Boolean network without fanin. The *primary outputs* (POs) are the nodes of the Boolean network without fanout. All other nodes in the Boolean network are *gates*.

6.3 Cut Rewriting

We have named our new Boolean rewriting algorithm *cut rewriting*. It can be applied directly to k-feasible Boolean networks. Its operation can roughly be described as consisting of two main stages. The first stage is largely reminiscent of the algorithms described in Chapter 5. We start by computing potential replacements subnetworks using exact synthesis. However, instead of eagerly rewriting the Boolean network as we did in Chapter 5, the replacement networks are stored in a conflict graph. A node of the conflict graph denotes a possible subnetwork replacement. Nodes are labeled with their node reduction gains. An edge between two nodes denotes a conflict between two replacements such that only one of them can be applied. In the second stage of cut rewriting, the conflict graph is used to determine a globally optimal subset of replacements by solving a maximum weighted vertex independent set problem (MWVIS). It is important note that, while we use exact synthesis to compute optimum replacement networks, the overall global optimization flow is still heuristic. The full pseudocode can be found in Algorithm 15. Note that some of the functions used in this pseudocode were previously defined in Section 5.2.2. In the remainder of this section we describe the steps shown there in greater detail.

The algorithm starts by computing all cuts for a cut size l and cut limit p. The cut size should be chosen relative to k. For example, l must be larger than k to find replacement candidates that lead to a gain, but if l is too large it can degrade efficiency of exact synthesis, as it is then required to synthesize functions that are too large. We have experimentally evaluated that for $k = 2$ and $k = 3$, cut sizes $l = 5$ and $l = 6$ lead to good results, respectively.

Once cut enumeration has concluded and we have used exact synthesis to find all replacement candidates, stage one has come to an end. Next, in stage two our task is to find a set of replacement candidates that maximize the overall gain. To this end, we use a graph $G = (V, E)$ as an additional data structure. It is initialized as the empty graph, and is iteratively constructed while enumerating replacement candidates for the cuts. Cuts are represented by the set of vertices V. There is an edge $(c, c') \in E$ if two cuts c and c' have overlapping logic. In that case they cannot be replaced simultaneously. Further, vertices $v \in V$ have weights w, such that $w(v)$ is the gain of a cut when replaced by its best found replacement network. Finally, r maps a vertex to the root node (in the subject graph) of the best replacement cut.

While executing stage one, the algorithm will have enumerated replacements (n', I) for each cut (n, I) using exact synthesis. In general, the replacements do not necessarily have to be size-optimum. In fact, relaxing the optimality constraint may improve SAT runtime. For further runtime control we can set thresholds on the conflict limit of the SAT solver [56]. For each replacement candidate the gain is computed using DryReplace and the best gain is stored in a variable gain together with the best replacement candidate in bestReplacement. If a replacement that leads to a gain can be found, then we add cut vertex to V and the mappings

Algorithm 15 The cut rewriting algorithm.

```
function CutRewrite(Boolean network N, cut size l, cut limit p)
    C ← CutEnumeration(N, l, p)
    G ← (V = ∅, E = ∅, w, r)
    for each gate n ∈ N do
        if MFFCSize(N, n) = 1 then
            continue
        end if
        for each leaves I ∈ C(n) do
            bestGain ← 0
            bestReplacement ← Λ
            for each replacement (n′, I) do
                gain ← DryReplace(N, n ↦ n′, I)
                if gain > bestGain then
                    bestGain ← gain
                    bestReplacement ← n′
                end if
            end for
            if bestReplacement ≠ Λ then
                Add vertex v = (n, I) to V
                w(v) ← bestGain
                r(v) ← bestReplacement
            end if
        end for
    end for
    for each node pair n1, n1 ∈ N, n1 ≠ n2 do
        for each I1 ∈ C(n1) and I2 ∈ C(n2) do
            if Cover(n1, I1) ∩ Cover(n2, I2) ≠ ∅ then
                Add edge (n1, I1) — (n2, I2) to E
            end if
        end for
    end for
    V′ ← MaximalIndependentVertexSet(G)
    for each (n, I) ∈ V′ do
        Replace(N, n ↦ r(n), I)
    end for
end function
```

w and r are updated with the gain and the replacement candidate, respectively. Afterwards, edges are added to E for each two cuts that have overlapping covers.

Finally, we need to select a good subset of non-conflicting replacement candidates. In order to do so, we heuristically solve the maximum weighted independent vertex set problem on G with respect to weights w. We use the greedy GWMIN algorithm [115], which provides an approximation guarantee of finding a solution with a weight of at least $\frac{1}{\Delta}\alpha(G)$, where Δ is the degree of G and $\alpha(G)$ is weight of the exact solution.

6.3.1 Efficiency Tricks & Don't Cares

Here, we consider two simple techniques that we can apply in order to improve the efficiency of our algorithm. First, we can skip all nodes whose MFFC contains just a single gate. Replacing such nodes cannot lead to any positive gain. Recall that Section 5.2.2 shows we can define a function to find a node's MFFC efficiently. Second, we can apply a caching technique similar to the one described in Chapter 5. The computation of replacement subnetworks in stage 1 uses exact synthesis. Hence, this stage can be implemented as described in Section 5.5 and Section 5.6, where we build up an NPN database on-demand. In doing so, we retain the advantages of on-the-fly synthesis while greatly reducing runtime for functions that we encounter more than once. When calling cut rewriting repeatedly, successive runs need to call exact synthesis only on new cut functions. Note that, when we use an NPN database in this way, cut rewriting can be viewed as a generalization of the optimization method described in Chapter 5, since 3-LUT networks include XMGs. The only caveat here is that the method from Chapter 5 uses a different subnetwork selection strategy. However, we can simplify the selection strategy used by cut rewriting. For example, we can choose to no longer solve the MWVIS problem. Doing so reduces cut rewriting to a generalization of the XMG size optimization method which can be applied to arbitrary k-feasible Boolean networks.

A complication arises when we want to support synthesis of partially specified functions, i.e. functions with don't care conditions. Supporting don't cares is desirable, because it allows us to use flexibilities caused by the logic network structure as additional optimization opportunities. However, for every n-input function, there are 2^n possible don't care patterns that we might encounter, since every entry in the truth table is either a care or a don't care bit. Thus, don't cares greatly increase the number of possible functions that we might encounter in the optimization flow. Moreover, it is not clear how to map a partially specified function to a unique NPN class, since NPN classes are fully specified. In fact, this remains an open problem at the time of writing. Due to this problem, we cannot construct the object S for partially specified functions. Therefore, our algorithm does not currently support the caching of partially specified functions. Hence, when applying cut rewriting with don't care computation enabled, we do not make use of the NPN database. Fortunately, don't care conditions do

provide extra degrees of freedom, and allow for smaller circuits. This makes partially specified functions easier to synthesize, which partially offsets the no-caching runtime penalty.

6.4 Experiments

We have implemented cut rewriting in the C++-17 *mockturtle* library.[1] *Mockturtle* is an open source logic network library. It is part of the EPFL logic synthesis libraries, which comprise a larger collection of open source EDA libraries [134]. Internally, our implementation uses *percy* for exact synthesis. Please refer to Appendix A for more details about percy, including code examples. By combining these EPFL libraries in a generic way, our implementation can, in principle, be applied to any k-LUT network. Our experiments indicate that using the current implementation, practical and scalable results can be achieved. Example scripts to recreate the experimental results are available in the *mockturtle* documentation.

In our experiments, we apply cut rewriting to improve the size of 3-LUT and 4-LUT networks for the combinational instances in the ISCAS benchmarks and the arithmetic instances in the EPFL benchmarks [5]. The baseline networks are obtained by performing a LUT mapping using '`&if -K k`' with $k \in \{3,4\}$ in ABC [23], respectively. In case of the EPFL benchmarks, we chose the best-known size-optimized 6-LUT benchmarks as a starting point.[2] As state-of-the-art area optimization we apply a synthesis script that interleaves priority-cut-based LUT mapping ('`&if`') [98], structural choices ('`&dch`' and '`&synch2`') [29, 96], and Boolean network optimization and resynthesis ('`&mfs`') [99]. We apply the synthesis script

```
&st; &synch2; &if -m -a -K k; &mfs -W 10;
  &st; &dch; &if -m -a -K k; &mfs -W 10
```

with the respective k parameter ten times and pick the best result that was encountered during all iterations. This optimization method is called *MFS* in the remainder.

To compare to this state-of-the-art area optimization script in ABC, we call the proposed cut rewriting algorithm repeatedly until no further gain in area can be achieved. We apply this optimization both on the baseline and on the networks obtained by MFS. We enable the use of don't care optimizations, although we use only *satisfiability don't cares* here. The use of don't cares allows us to find more optimization opportunities. However, it also precludes us from building an NPN database. This, in turn, increases the runtime of our algorithm.

Tables 6.1 and 6.2 show the experimental results for 3-LUTs and 4-LUTs, respectively. The table lists the baseline, the results obtained after MFS, the results obtained after cut rewriting,

[1] https://github.com/lsils/mockturtle
[2] See https://lsi.epfl.ch/benchmarks

Table 6.1 – Cut rewriting experimental results for 3-LUT resynthesis

Name	Baseline				MFS			MFS + Cut rewriting			Improvement
	PIs	POs	gates	levels	gates	levels	time (s)	gates	levels	time (s)	
c432	36	7	113	16	71	19	0.66	**68**	22	1.17	4.23%
c499	41	32	112	9	102	9	2.27	102	9	0.51	0.00%
c880	60	26	175	13	141	14	1.87	**139**	14	2.65	1.42%
c1355	41	32	112	9	102	9	1.79	102	9	0.55	0.00%
c2670	157	64	304	11	216	12	2.02	**211**	14	2.32	2.31%
c3540	50	22	563	19	316	20	5.33	**309**	20	9.16	2.22%
c5315	178	123	838	14	521	15	5.89	**510**	15	12.01	2.11%
c6288	32	32	733	31	748	33	30.81	748	33	0.02	0.00%
c7552	207	108	666	14	540	32	6.12	**522**	34	17.32	3.33%
adder	256	129	827	67	428	85	4.92	**256**	128	1.08	40.19%
bar	135	128	1018	7	896	7	10.13	896	7	0.00	0.00%
div	128	128	13202	2299	8465	2170	136.41	**7010**	2290	4030.98	17.19%
log2	32	32	21759	216	15927	201	588.34	**15146**	195	22650.17	4.90%
max	512	130	891	249	823	249	9.56	**808**	251	5.25	1.82%
multiplier	128	128	18983	147	11346	141	366.53	**11196**	145	9771.94	1.23%
sin	24	25	4334	99	2989	102	907.73	**2851**	99	308.27	4.62%
sqrt	128	64	12918	2116	8031	2147	124.13	**7010**	2174	3115.73	12.71%
square	64	128	15290	168	6931	165	446.19	**6789**	166	188.10	2.05%
Average											5.58%
Sum							2650.00			40117.26	

Table 6.2 – Cut rewriting experimental results for 4-LUT resynthesis

Name	Baseline				MFS			MFS + Cut rewriting			Improvement
	PIs	POs	gates	levels	gates	levels	time (s)	gates	levels	time (s)	
c432	36	7	100	10	52	16	1.52	52	16	0.19	0.00%
c499	41	32	78	5	78	6	5.17	78	6	0.02	0.00%
c880	60	26	125	9	108	14	4.14	**106**	14	0.29	1.86%
c1355	41	32	78	5	80	6	5.74	80	6	0.21	0.00%
c2670	157	64	204	7	178	9	7.98	**161**	9	0.26	9.55%
c3540	50	22	348	12	236	16	14.70	**231**	16	1.69	2.12%
c5315	178	123	506	10	425	12	17.28	**383**	12	1.14	9.88%
c6288	32	32	503	25	494	31	89.78	494	31	0.10	0.00%
c7552	207	108	520	8	427	24	17.40	**424**	24	1.11	0.70%
adder	256	129	529	44	341	84	15.96	**298**	127	0.06	12.60%
bar	135	128	1018	7	896	7	35.98	896	7	0.00	0.00%
div	128	128	9597	1486	5113	2007	390.33	**4681**	2069	27.28	8.45%
log2	32	32	14021	128	11659	172	1993.53	**10761**	166	14097.98	7.70%
max	512	130	1074	135	785	245	27.62	**784**	245	1.02	0.13%
multiplier	128	128	11256	98	8264	138	1223.32	**8084**	137	1893.17	2.61%
sin	24	25	2921	62	2172	88	4393.63	**2056**	87	60.49	5.34%
sqrt	128	64	9139	1386	5004	1945	384.45	**4534**	1992	30.46	9.21%
square	64	128	8843	104	5737	132	1153.30	**5588**	140	35.44	2.60%
Average											4.04%
Sum							9781.84			16150.65	

and the results obtained by applying cut rewriting after MFS. For each it lists the number of gates and the number of logic levels. It also lists the runtime in seconds. In case of *MFS + Cut rewriting* it only lists the additional time required by cut rewriting. The last column shows the improvement that can be obtained by calling cut rewriting on the results already optimized by MFS. The cut size and cut limit for cut enumeration are $l = 6$ and $p = 12$, respectively. We compute one replacement candidate for each cut using exact synthesis with a conflict limit of 1000.

In a direct comparison, cut rewriting cannot achieve the quality of MFS for most of the benchmarks. It also requires more runtime in total. However, for the benchmarks *adder*, *div*, and *sqrt*, cut rewriting leads to better results. For the cases *c6288* and *bar*, the quality results are the same, but cut resynthesis finds them more quickly.

The strength of cut resynthesis becomes evident when it is used as a post-optimization method *after* MFS has iterated repeatedly to find the best possible network. We see that cut rewriting can find additional improvements on top of the highly optimized networks found by MFS. Often this improvement is requires a comparably small runtime overhead. The average improvement is 5.58% and 4.04% when resynthesizing 3-LUT and 4-LUT networks, respectively. The best improvement is achieved for the 128-bit adder, which improved by 40.19% when considering 3-LUT networks. Starting from a baseline implementation that has 67 logic levels it manages to regain the size-optimal carry ripple implementation with one sum gate (XOR-3) and one carry gate (majority-3) for each pair of input bits.

6.5 Summary

In this chapter, we have presented a generic DAG-aware rewriting algorithm which can be applied directly to k-feasible Boolean networks. It can be viewed as an alternative to, or generalization of, the XMG-based optimization method from Chapter 5. Our method combines the computation of local rewriting gains with a global view of the subject graph. We achieve this by, not applying replacements in an ad-hoc or greedy way, but instead by storing all possible local rewriting gains in a conflict graph. Nodes in the conflict graph correspond to, possibly overlapping, cones of logic in the subject graph. Edges in the graph specify which subnetworks cannot be rewritten simultaneously. Thus, we can use the computed gains and conflicts to maximize the overall achievable rewriting gain. We do so by computing an approximate solution to the MWVIS problem. Our algorithm retains the desirable features of XMG-based optimization, including support for an NPN database. Moreover, we support partially specified functions, which allows us to take advantage of additional optimizations which are warranted by don't care conditions. Using our new algorithm, we show size improvements up to **40.19%** and **12.60%** when resynthesizing heavily optimized 3-LUT networks and 4-LUT networks, respectively.

7 Conclusions

In this thesis, we have investigated the problem of finding optimum circuits for Boolean functions. As we have seen in Section 2.7, while it is not known if this problem is computationally intractable, it is unlikely that efficient (polynomial-time) algorithms exist. Indeed, the existence of such algorithms would have radical implications for computational and circuit complexity theory. In other words, there is plenty of theoretical evidence to suggest that the exact synthesis problem for multi-level logic networks is very hard.

Faced with this difficult problem, we have turned to SAT for a solution. Although SAT is NP-complete, we also know that the efficient search procedures of modern SAT solvers manage to solve problems with millions of variables. Indeed, the incredible progress made by SAT solvers over the past decades can be taken as proof that many problems that we care about can be solved in practice, even if they require an exponential amount of time in the worst case. There is an analogy with circuit complexity here. We know that, in the limit, most Boolean functions require circuits with an exponential number of gates. However, in practice we have found a great many useful circuits that use relatively few gates. Indeed, if that had not been the case, the information revolution would not have been possible, and computers would not be practical machines today.

Given the apparent intractability of exact synthesis, our goal in this work was not to find a polynomial-time algorithm. Rather, we have taken the approach suggested by the recent success of SAT. By encoding into CNF, we are able to solve many problems that are of both theoretical and practical interest. Thus, the contributions we have made here serve to push the "knee of the exponential" further to the right, as shown in Figure 7.1. In other words, by improving SAT-based exact synthesis algorithms, we can ensure that we encounter worst-case runtimes with larger problem instances, and that we encounter them less frequently. In practice, this allows us to synthesize circuits more quickly, as well as find circuits for larger functions. From another point of view, our contributions extend the practical applications in

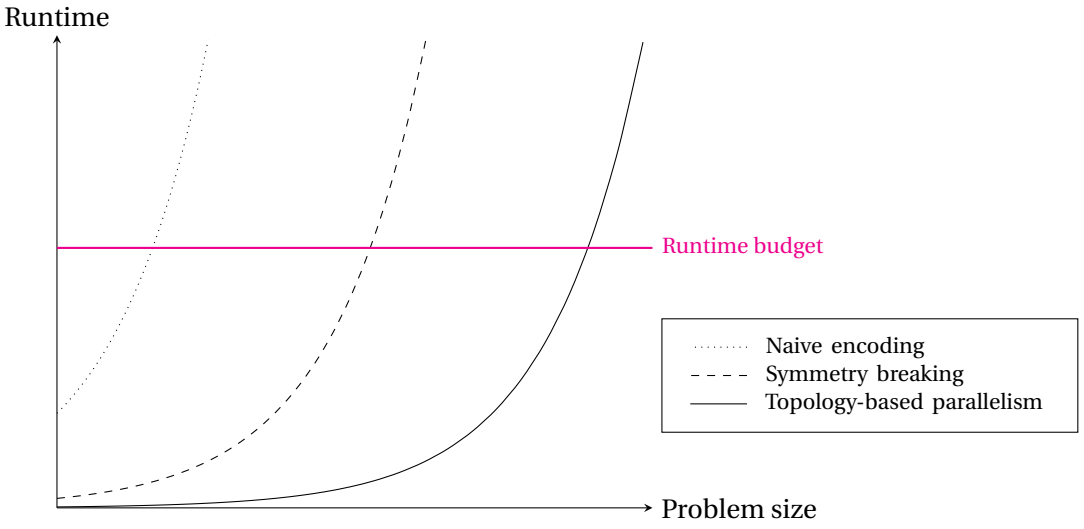


Figure 7.1 – Even without an exponential speedup, we can make SAT-based exact synthesis more practical through techniques such as symmetry breaks and topology-based parallelism.

which SAT-based exact synthesis may be used, given some fixed upper bound on our runtime budget. In summary, this work should not be viewed as an attempt to find fundamentally new bounds on the MCSP problem, or to reduce its complexity by some exponential factor. Rather, it is a work of engineering in which we analyze and propose new techniques to manage the seemingly intractable nature of this problem.

7.1 Thesis Contributions

Here, we give a brief summary of the contributions made in this work, in the order that they have been presented.

- **Encoding analysis and comparison.** We have contrasted and compared different CNF encodings of the exact synthesis problem. From a theoretical point of view, we have analyzed their structure, including the different types of clauses and numbers of structural variables they require. From a experimental point of view, we have measured their runtime behavior on a number of benchmarks. Our experiments show that choosing the proper combination of encoding and symmetry breaks can make a great difference. Any project in which exact synthesis is to be applied should therefore ensure that it selects a good combination for the expected distribution of problem instances.
- **Introducing DAG topology families.** We have introduced the notion of families of DAG topologies for SAT-based exact synthesis. Specifically, we give two concrete examples

of different topology family types, which we have named *fences* and *partial DAGs*, respectively. We contrast these topology families with regular DAGs, and show how the number of family members is orders of magnitude smaller than the number of DAGs. This makes synthesis based on these topology types feasible, as enumerating them can be done quickly, whereas this is not the case for regular DAGs. We confirm, in several experiments, that taking advantage of the proper topology families can significantly reduce synthesis runtime.

- **Parallel exact synthesis through DAG topology families.** Many logic synthesis algorithms suffer from the fact that they are difficult to parallelize. This has also been the case for SAT-based exact synthesis. We show how DAG topology families can be used to unlock embarrassingly parallel synthesis algorithms. Moreover, we show that this kind of domain-specific parallelism can significantly outperform the generic parallelism offered by state-of-the-art parallel SAT solvers.

- **Exact synthesis for function classification.** The first application we discuss is the theoretical task of classifying functions in terms of their combinational complexity. We show how a combination of techniques including NPN classification and exact synthesis can be used this efficiently.

- **Novel basis for logic rewriting.** We have introduced the XMG data structure as a new compact logic representation. Using XMGs as our representation, we propose a novel logic rewriting architecture based on an on-demand combination of exact synthesis, NPN classification, and memoization. Many conventional rewriting algorithms require some notion of precomputation, which becomes intractable for larger functions. We show that, by computing optimum networks only for those function classes we encounter in practice, we can avoid such a precomputation step. Moreover, our dynamic use of exact synthesis allows us to rewrite larger cones of logic, thus unlocking significant improvements over the state-of-the-art.

- **Generic Boolean network rewriting.** Generalizing and improving on our work with XMGs, we have introduced a generic rewriting framework for Boolean networks. This new algorithm, which we have named *cut rewriting*, combines local rewriting improvements made by exact synthesis with a global view of the subject graph. The global view is represented by a a conflict graph which contains those cones of logic that cannot be rewritten concurrently. By solving the weighted maximum independent vertex set problem on the conflict graph, we find a valid set of replacement subnetworks which maximizes the overall global size reduction.

- **Software package.** In order to perform the experiments presented in this work, we have developed the open source *percy* library. Part of the larger collection of EPFL logic synthesis libraries, It is a header-only C++ package which implements all the synthesis

algorithms described in this thesis. Details about the design and implementation of *percy* can be found in Appendix A.

7.2 Future Work and Open Problems

With a problem as hard as exact synthesis, it is difficult for any one work to close the door on all of its aspects. Indeed some interesting avenues for future work as well as some open problems remain, and we discuss some of them here.

- We have seen that choosing the proper combination of encoding and symmetry breaks greatly impacts synthesis runtime. The heuristics used by the SAT solver can be another significant factor. Currently, not much is known about exactly how these different factors affect runtime. Consider the following questions. Given a set of benchmarks, how do we choose the proper encoding, symmetry breaks, and SAT solver? Is there any theoretically principled way of doing so? Currently, the best way to configure our exact synthesis system is by sampling from the benchmark distribution. This gives us an idea of the expected performance of our system. However, determining expected performance in such an experimental way is essentially a hyper-parameter auto-tuning optimization. This may take a very long time, and is not always feasible. At the time of writing, there is no theoretical framework that we can use to answer questions of this kind.

- An interesting unexplored research direction is the development of domain-specific SAT solvers that specialize in solving exact synthesis problems. One can imagine that such solvers have much faster constraint propagation and variable heuristics due to the fact that they can take advantage of knowing, in advance, the structure of the problem they are solving. For example, so-called CIRCUIT-SAT solvers have already been successfully applied to logic synthesis algorithms [74, 90].

- We know that choosing proper DAG topologies, whether these are proper DAGs or topology families, can significantly reduce synthesis runtime. An open problem is to search for, or generate, DAG topologies that are "good" in some sense. For example, we know that some DAGs are more expressive, in the sense that more functions can be represented by some topologies than others. Can we find some features that characterize what it means for a particular DAG topology to be expressive? Or perhaps it is possible to construct a conditional probability distribution which tells us, given some function f, what types of topologies we can likely use to represent f.

- Our contributions suggest a new direction for logic rewriting algorithms. Such new algorithms can take advantage of two novel techniques we have described. First, they can exploit topology-based parallelism, which significantly speeds up the synthesis,

and allows us to synthesize larger functions. Second, they can make use of distributed computing architectures, as described in Chapter 5. In such architectures, functions are synthesized, stored, and retrieved in an on-demand fashion. Designing such systems is an nontrivial task which will require careful engineering, experimentation, and the weighing of many trade-offs. But from this we can see that our contributions neatly fit into a new type of logic synthesis system architecture, which has also been described as EDA 3.0 [140].

- Recently, there has been substantial interest in applying machine learning algorithms in EDA [14, 148]. There are opportunities for exact synthesis in this domain as well. For example, one might use exact synthesis to find optimum *ground truth* logic networks. These can then be used as training data for algorithms that learn how to optimize logic networks [55].

7.3 Final Remarks

We have analyzed different algorithms for SAT-based exact synthesis for multi-level logic networks. We have proposed novel core synthesis algorithms, and shown how they can be applied to problems that are of both practical and theoretical interest. Finally, we have introduced a novel embarrassingly parallel synthesis method which can unlock the full computational potential of today's highly parallel and distributed computing systems.

Although it remains a very difficult problem, the contributions made in this work are a significant step in the right direction, and show how we can design new practical systems based on this important algorithm. SAT-based exact synthesis is ready for the era of parallel and distributed computing, and is fit to be included in the pantheon of future EDA algorithms.

A *percy*: an exact synthesis library

percy is one of the EPFL logic synthesis libraries [134]. These are a collection of modular open source C++ libraries for the development of logic synthesis applications. Each of them targets a specific aspect of logic synthesis, such as ESOP minimization, generic multi-level logic network optimization, or fast truth table operations. Their designs are purposefully lightweight and modular, so that larger synthesis tools can be built by composing them. *percy* itself provides various SAT based exact synthesis engines. These include all the methods described in this thesis, such as the conventional monolithic encodings and the parallel topology-based synthesis methods. The constraints and algorithms of such synthesis engines may be quite dissimilar. Moreover, it is not always obvious which combination will be superior in a specific domain. It is often desirable to experiment with several methodologies and solving backends to find the right fit. The aim of *percy* is to provide a flexible common interface that makes it easy to construct a parameterizable synthesis engine suitable for different domains.

percy also serves as an example of the design philosophy behind the EPFL libraries. It is built on top of *kitty* – a truth table library from the EPFL collection – which it uses to construct synthesis specifications. Thus, it shows how the lightweight logic synthesis libraries can be easily composed to build up ever more complex structures.

In this appendix we describe the design choices behind *percy*. Although is a lightweight library, *percy* provides many functions, including features such as (partially constrained) solution enumeration, and topology-based parallel synthesis. Therefore, our goal here is not describe *percy* in its entirety, but rather to provide a high-level overview, accompanied by some simple code examples. Together, these should demonstrate how *percy* makes it easy to construct different synthesis engines. In order to run the examples one can find the library source code at https://github.com/whaaswijk/percy. For the interested reader, this repository also contains many code examples of some of *percy*'s more advanced features.

Synthesis using *percy* involves four main components:

1. *Specifications* – Specification objects contain the information essential to the synthesis process such as the functions to synthesize, I/O information, and a number of optional parameters such as conflict limits for time-bound synthesis, or topology information.
2. *Encoders* – Encoders are objects which convert specifications to CNF formulas. There are various ways to create such encodings, and by separating their implementations it becomes simple to use encodings in different settings.
3. *Solvers* – Once an encoding has been created, we use a SAT solver to find a solution. Currently supported are ABC's `bsat` and `bmcg` solvers, the Glucose and Glucose-Syrup solvers, and the CryptoMinisat solver [46, 13, 138]. Adding a new SAT solver to *percy* is as simple as declaring a handful of interface functions.
4. *Chains* – Boolean chains are the result of exact synthesis.

A typical exact synthesis workflow will have some source for generating specifications. These specifications are then given to an exact synthesis method which converts them into optimum Boolean chains. Internally, the synthesis methods will compose its underlying encoder and SAT solver in a specific synthesis flow. For example, a resynthesis algorithm might generate cuts in a logic network which serve as specifications. These may then be fed to a synthesis engine, which may internally use a monolithic encoding with the `bsat` solver, or a topology-based encoding with the Glucose-Syrup solver. If the resulting optimum Boolean chains leads to an improvement, are replaced in the logic network. In optimizing this workflow, *percy* makes it easy to swap out one synthesis engine for another, to change CNF encodings, or to switch to a different SAT solver.

A.1 Code Examples

The example in Listing A.1 shows how *percy* can be used to synthesize an optimum full adder. While simple, it demonstrates some common interactions between the various components. We see that first a specification object is created, which is then passed to the default synthesis engine using the `synthesize` function. The result is an optimum Boolean chain using a full adder.

Listing A.1 – Synthesizing a full adder with *percy*.

```
#include <percy/percy.hpp>
using namespace percy;

...

// We use a specification object to describe what functions we want
// to synthesize, as well as any other constraints (e.g. fanin size)
spec spec;
spec.set_nr_out(2); // Synthesize 2 functions.

// The result of synthesis is a Boolean chain
// (i.e. a multi-level logic network)
chain c;

// The sum and carry functions represent the outputs of the chain
// that we want to synthesize. In percy, functions are represented
// by truth tables, using the kitty library.
dynamic_truth_table x(3), y(3), z(3);

create_nth_var(x, 0);
create_nth_var(y, 1);
create_nth_var(z, 2);

const auto sum = x ^ y ^ z;
const auto carry = ternary_majority(x, y, z);
spec[0] = sum;
spec[1] = carry;

// Call percy with the specification we've constructed.
auto result = synthesize(spec, c);

// Ensure that synthesis was successful.
assert(result == success);

// Simulate the synthesized circuit and ensure that it
// computes the correct functions.
auto sim_fs = c.simulate();
assert(sim_fs[0] == sum);
assert(sim_fs[1] == carry);
```

By default, *percy* synthesizes chains with 2-input steps, using a monolithic SSV encoding and the `bsat` solver. It also enables all symmetry breaks. As we have seen above, this is not necessarily the most efficient engine for all problems. Suppose therefore that this particular combination is not suitable for our particular workflow. We can then easily switch to a new engine by changing just a few lines of code. Consider the example in Listing A.2. It achieves the same task but now using the MSV encoding and the parallel Glucose-Syrup solver. Moreover, it does not apply the co-lexicographical ordering symmetry break. Note that while we now use a completely different synthesis engine, we only changed four lines of code.

Listing A.2 – Creating an engine with the MSV encoding and Glucose-Syrup solver.

```
#include <percy/percy.hpp>
using namespace percy;

// We now want to use the Glucose-Syrup solver
glucose_wrapper solver;

// And the MSV encoding
msv_encoder encoder(solver);

spec spec;
spec.set_nr_out(2);
// Do not impose co-lexicographical ordering on the steps
spec.add_colex_clauses = false;

chain c;
dynamic_truth_table x(3), y(3), z(3);
create_nth_var(x, 0);
create_nth_var(y, 1);
create_nth_var(z, 2);

const auto sum = x ^ y ^ z;
const auto carry = ternary_majority(x, y, z);
spec[0] = sum;
spec[1] = carry;

// Synthesize using the new engine
auto result = synthesize(spec, c, solver, encoder);

assert(result == success);

auto sim_fs = c.simulate();
assert(sim_fs[0] == sum);
assert(sim_fs[1] == carry);
```

Next, Listing A.3 shows how *percy* can be used to perform topology-based synthesis. In particular, it shows an alternative approach to the same example, but now with a fence-based synthesis engine using the SSV encoding (adapted to fence-based synthesis) and the `bmcg` solver. Again, we have to make only minor changes in order to construct a completely different engine.

Listing A.3 – Fence-based synthesis of a full adder.

```
#include <percy/percy.hpp>
using namespace percy;

// We now use the bmcg solver
bmcg_wrapper solver;

// And a fence-based encoding
ssv_fence_encoder encoder(solver);

spec spec;
spec.set_nr_out(2);
chain c;
dynamic_truth_table x(3), y(3), z(3);
create_nth_var(x, 0);
create_nth_var(y, 1);
create_nth_var(z, 2);

const auto sum = x ^ y ^ z;
const auto carry = ternary_majority(x, y, z);
spec[0] = sum;
spec[1] = carry;

// Synthesize using the new engine
auto result = fence_synthesize(spec, c, solver, encoder);

assert(result == success);

auto sim_fs = c.simulate();
assert(sim_fs[0] == sum);
assert(sim_fs[1] == carry);
```

Finally, we give an example of how one might use topology-based parallelism in *percy*. Te script shown in Listing performs an experiment similar to the one presented in Section 3.10. It synthesizes a list of functions using both the parallel Glucose-Syrup MultiSolvers solver backend, as well as partial DAG synthesis. It allocates 24 threads for both synthesis methods,

and prints a running average of the required synthesis time as it processes the list. This is another example of how *percy* makes it easy to compare different methodologies.

Listing A.4 – Example of topology-based parallelism in *percy*.

```
#include <percy/percy.hpp>
...

chain c;
spec spec;
glucose_wrapper solver; // Create MultiSolvers instances
ssv_encoder encoder(solver);

...

const auto nr_threads = 24;
int multisolvers_elapsed = 0;
int pd_elapsed = 0;
solver.set_nr_threads(nr_threads);
int ctr = 1;
for (const auto& tt : functions) {
  spec[0] = tt;

  auto begin = steady_clock::now();
  auto res = pd_synthesize_parallel(spec, c, nr_threads); // Partial DAG synth
  auto end = steady_clock::now();
  auto elapsed_time = duration_cast<milliseconds>(end - begin).count();
  pd_elapsed += elapsed_time;
  assert(res == success);

  begin = steady_clock::now();
  res = synthesize(spec, c, solver, encoder);
  end = steady_clock::now();
  elapsed_time = duration_cast<milliseconds>(end - begin).count();
  multisolvers_elapsed += elapsed_time;
  assert(res == success);

  // Print running averages
  printf("Iteration %d\n", ctr);
  printf("Average synth time (MULTI): %.2fms\n",multisolvers_elapsed/(1.0*ctr));
  printf("Average synth time (PD): %.2fms\n",pd_elapsed/(1.0*ctr));

  ctr++;
}
```

A.2 A Note on Correctness

When synthesizing networks using *percy*, it is essential that we be able to rely on its results. After all, when we generate theoretical results such as those in Chapter 4, we must be able to trust that they are indeed correct. This is also the case in more practical applications, such as logic rewriting. When optimizing a logic network, any error in the synthesis of a subnetwork could invalidate the entire subject graph. Thus, correct synthesis is paramount. However, verification of an advanced logic synthesis algorithm is nontrivial.

To verify the results that *percy* generates, we must take into account two distinct dimensions of verification: (i) we must ensure that synthesized chains are functionally correct, and (ii) we must ensure that synthesized chains are indeed optimum chains . We verify this in four (partially) orthogonal ways. To verify (i) we do the following:

1. We simulate synthesized chains to ensure that they compute the correct functions. Simulating a Boolean chain is a relatively simple algorithm, as it reduces to evaluating the chain's outputs on all minterms. Due to this simplicity, it is unlikely that the outcome of simulation is incorrect. Hence, if the results of simulation tells us that a chain implements the correct function, this significantly increases our confidence.

2. We incorporate *percy* as a core part of larger algorithms. If, when applied in this larger context, we still achieve functionally correct results, this further increases our confidence in *percy*. For example, suppose we perform logic rewriting with *percy*. If the optimized circuits are functionally correct – which we can verify using an external tool such as a formal equivalence checker – then it becomes less likely that *percy* produces incorrect chains.

Next, to verify (ii), we do the following:

3. We can use *percy* to construct tables which show the minimum size for functions, mapping the number of gates to the number of functions with that number as minimum size. We can then verify these tables against results that can be found in other works, such as that by Knuth [69]. From these tables, we can see that our algorithms indeed find the optimum sizes, otherwise the tables are unlikely to match.

4. Unfortunately, point (3) can only be achieved for functions of 4 or 5 inputs, since no such tables exist for larger input sizes. To further verify our implementations, we have also compared them to existing exact synthesis algorithms, such as those found in ABC. By comparing to these trusted implementations, we again have greater assurance that ours are correct.

By applying the verification processes outlined above, we can be reasonably sure that *percy* functions correctly. Note that the difficulty we face here is similar to that of verifying SAT solvers. Barring formal verification of a SAT solver, the best we can do is to test it against known results and other trusted solver implementations.

Bibliography

[1] Afshin Abdollahi and Massoud Pedram. A new canonical form for fast Boolean matching in logic synthesis and verification. In *Proceedings of the Design Automation Conference (DAC)*, pages 379–384, 2005.

[2] Sheldon B. Akers. Synthesis of combinational logic using three-input majority gates. In *Foundations of Computer Science*, pages 149–157, 1962.

[3] Sheldon. B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27(6): 509–516, 1978.

[4] Eric Allender, Joshua A. Grochow, Dieter van Melkebeek, Cristopher Moore, and Andrew Morgan. Minimum Circuit Size, Graph Isomorphism, and Related Problems. *SIAM Journal on Computing*, 47(3), 2018. doi: 10.1137/17M1157970.

[5] Luca G. Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. Boolean logic optimization in majority-inverter graphs. In *Proceedings of the Design Automation Conference (DAC)*, pages 1–6, 2015.

[6] Luca G. Amarù, Pierre-Emmanuel Gaillardon, Anupam Chattopadhyay, and Giovanni De Micheli. A Sound and Complete Axiomatization of Majority-n Logic. *IEEE Trans. Computers*, 2016.

[7] Luca G. Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. Majority-Inverter Graph: A novel data-structure and algorithms for efficient logic optimization. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014. doi: 10.1145/2593069.2593158.

[8] Luca G. Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. Boolean logic optimization in majority-inverter graphs. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015. doi: 10.1145/2744769.2744806.

[9] Luca G. Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. Majority-inverter graph: A new paradigm for logic optimization. volume 35, pages 806–819, May 2016. doi: 10.1109/TCAD.2015.2488484.

[10] Luca G. Amarù, Mathias Soeken, Patrick Vuillod, Jiong Luo, Alan Mishchenko, Pierre-Emmanuel Gaillardon, Janet Olson, Robert K. Brayton, and Giovanni De Micheli. Enabling exact delay synthesis. In *Proceedings of the IEEE/ACM Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 352–359, 2017. doi: 10.1109/ICCAD.2017.8203799.

[11] Robert L. Ashenhurst. The Decomposition of Switching Functions. *Proceedings of the International Symposium on the Theory of Switching*, pages 74–116, 1957.

[12] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 399–404, 2009.

[13] Gilles Audemard and Laurent Simon. Glucose and Syrup in the SAT Race 2015. In *Reports on the SAT 2015 Competition*, 2015.

[14] Peter A. Beerel and Massoud Pedram. Opportunities for Machine Learning in Electronic Design Automation. In *Proceedings of the Int'l Symposium on Circuits and Systems (ISCAS)*, Florence, Italy, 5 2018.

[15] Luca Benini and Giovanni De Micheli. A survey of boolean matching techniques for library binding. *ACM Trans. Design Autom. Electr. Syst.*, 2(3):193–226, 1997.

[16] Luca Benini and Giovanni De Micheli. Networks on chips: A new soc paradigm. *Computer*, 35(1):70–78, January 2002. ISSN 0018-9162. doi: 10.1109/2.976921. URL https://doi.org/10.1109/2.976921.

[17] Luca Benini, Patrick Vuillod, and Giovanni De Micheli. Iterative Remapping for Logic Circuits. *TCAD*, 17(10):948–964, 1998.

[18] Anna Bernasconi, Valentina Ciriani, Rolf Drechsler, and Tiziano Villa. Logic Minimization and Testability of 2SPP Networks. *IEEE TCAD*, 27(7):1190–1202, 2008. doi: 10.1109/DSD.2009.131.

[19] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfability*. IOS Press, 2009. ISBN 978-1-58603-929-5.

[20] Garrett Birkhoff and Stephen Anthony Kiss. A ternary operation in distributed lattices. *Bull. of the Amer. Math. Soc.*, pages 749–752, 1947.

[21] Norbert Blum. A Boolean function requiring $3n$ network size. *Theor. Comput. Sci.*, 28: 337–345, 1984.

[22] Robert K. Brayton and C. McMullen. The Decomposition and Factorization of Boolean Expressions. In *Proceedings of the Int'l Symposium on Circuits and Systems (ISCAS)*, pages 49–54, 1982.

[23] Robert K. Brayton and Alan Mishchenko. ABC: an academic industrial-strength verification tool. In *Computer Aided Verification*, pages 24–40, 2010.

[24] Robert K. Brayton, Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Boston, Massachusetts, 1984. ISBN 9781461328216.

[25] Robert K. Brayton, Richard L. Rudell, Alberto L. Sangiovanni-Vincentelli, and Albert R. Wang. Mis: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(6):1062–1081, November 1987. ISSN 0278-0070. doi: 10.1109/TCAD.1987.1270347.

[26] Robert K. Brayton, Gary D. Hachtel, and Alberto L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, 1990.

[27] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-decision Diagrams. *ACM Comput. Surv.*, 24(3):293–318, September 1992. ISSN 0360-0300. doi: 10.1145/136035.136043. URL http://doi.acm.org/10.1145/136035.136043.

[28] Satrajit Chatterjee. *On Algorithms for Technology Mapping*. PhD thesis, University of California at Berkeley, 2007.

[29] Satrajit Chatterjee, Alan Mishchenko, Robert K. Brayton, Xinning Wang, and Timothy Kam. Reducing Structural Bias in Technology Mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(12):2894–2903, Dec 2006. ISSN 0278-0070. doi: 10.1109/TCAD.2006.882484.

[30] Deming Chen and Jason Cong. DAOmap: a depth-optimal area optimization mapping algorithm for FPGA designs. In *Proceedings of the IEEE/ACM Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 752–759, 2004.

[31] Valentina Ciriani. Synthesis of SPP Three-Level Logic Networks Using Affine Spaces. 22 (10):1310–1323, 2003.

[32] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. *Counterexample-Guided Abstraction Refinement*, pages 154–169. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. ISBN 978-3-540-45047-4. doi: 10.1007/10722167_15.

[33] Martin Cohn and Richard Lindaman. Axiomatic Majority-Decision Logic. *IRE Trans. on Electronic Computers*, 10:17–21, 1961.

[34] Jason Cong and Yuzheng Ding. FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 13(1):1–12, 1994.

[35] Jason Cong and Yuzheng Ding. On area/depth trade-off in LUT-based FPGA technology mapping. *IEEE Trans. VLSI Syst.*, 2(2):137–148, 1994.

[36] Jason Cong and Yean-Yow Hwang. Simultaneous depth and area minimization in LUT-based FPGA mapping. In *Proceedings of the Int'l Symposium on Fied-Programmable Gate Arrays (FPGA)*, pages 68–74, 1995.

[37] Jason Cong, Chang Wu, and Yuzheng Ding. Cut Ranking and Pruning: Enabling a General and Efficient FPGA Mapping Solution. In *Proceedings of the Int'l Symposium on Fied-Programmable Gate Arrays (FPGA)*, pages 29–35, 1999.

[38] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM. doi: 10.1145/800157.805047.

[39] Olivier Coudert, Christian Berthet, and Jean C. Madre. Verification of Synchronous Sequential Machines Based on Symbolic Execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 365–373, New York, NY, USA, 1990. Springer-Verlag New York, Inc. ISBN 0-387-52148-8. URL http://dl.acm.org/citation.cfm?id=88032.88165.

[40] Maurizio Damiani and Giovanni De Micheli. Don't care set specifications in combinational and synchronous logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(3):365–388, March 1993. ISSN 0278-0070. doi: 10.1109/43.215001.

[41] Edward S Davidson. An Algorithm for NAND Decomposition Under Network Constraints. *IEEE Transactions on Computers*, C-18(12):1098–1109, 1969.

[42] Edward S. Davidson. An Algorithm for NAND Decomposition Under Network Constraints. *IEEE Trans. Computers*, 18(12):1098–1109, 1969.

[43] Michele De Marchi, Davide Sacchetto, Jian Zhang, Stefano Frache, Pierre-Emmanuel Gaillardon, Yusuf Leblebici, and Giovanni De Micheli. Top–down fabrication of gate-all-around vertically stacked silicon nanowire fets with controllable polarity. *IEEE Transactions on Nanotechnology*, 13(6):1029–1038, 2014.

[44] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[45] Niklas Eén. Practical SAT - a tutorial on applied satisfiability solving. In *FMCAD*, 2007.

[46] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. pages 502–518, 2003. doi: 10.1007/978-3-540-24605-3_37. URL https://doi.org/10.1007/978-3-540-24605-3_37.

[47] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing*, pages 502–518. Springer Berlin Heidelberg, 2004.

[48] Amir H. Farrahi and Majid Sarrafzadeh. Complexity of the lookup-table minimization problem for FPGA technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 13(11):1319–1332, 1994.

[49] Merrick L. Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984.

[50] Eiichi Goto and H. Takahasi. Some theorems useful in threshold logic for enumerating Boolean functions. In *International Federation for Information Processing Congress*, pages 747–752, 1962.

[51] Winston J. Haaswijk, Luca G. Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. NEM Relay Design with Biconditional Binary Decision Diagrams. In *IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, 2015.

[52] Winston J. Haaswijk, Mathias Soeken, Luca G. Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. LUT Mapping and Optimization for Majority-Inverter Graphs. In *Proceedings of the Int'l Workshop on Logic Synthesis (IWLS)*, 2016.

[53] Winston J. Haaswijk, Mathias Soeken, Luca G. Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. A Novel Basis for Logic Rewriting. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017.

[54] Winston J. Haaswijk, Eleonora Testa, Mathias Soeken, and Giovanni De Micheli. Classifying Functions with Exact Synthesis. In *ISMVL*, 2017.

[55] Winston J. Haaswijk, Edo Collins, Benoit Seguin, Mathias Soeken, Sabine Süsstrunk, Frédéric Kaplan, and Giovanni De Micheli. Deep Learning for Logic Synthesis Algorithms. In *Proceedings of the Int'l Symposium on Circuits and Systems (ISCAS)*, Florence, Italy, 5 2018.

[56] Winston J. Haaswijk, Alan Mishchenko, Mathias Soeken, and Giovanni De Micheli. SAT Based Exact Synthesis Using DAG Topology Families. In *Proceedings of the 55th Annual Design Automation Conference*, DAC'18, pages 53:1–53:6, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5700-5. doi: 10.1145/3195970.3196111. URL http://doi.acm.org/10.1145/3195970.3196111.

[57] Youssef Hamadi. ManySAT : a Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(5):245–262, 2009. doi: 10.1152/japplphysiol.00460.2010.

[58] Michael A. Harrison. The Number of Equivalence Classes of Boolean Functions Under Groups Containing Negation. *IEEE Transactions on Electronic Computers*, EC-12(5): 559–561, 1963. ISSN 0367-7508.

[59] Michael A. Harrison. *Combinatorial Problems in Boolean Algebras and Applications to the Theory of Switching*. PhD thesis, The University of Michigan, 1963.

[60] Johan Håstad. Almost optimal lower bounds for small depth circuits. In *Annual ACM Symposium on Theory of Computing*, pages 6–20, 1986.

[61] William Hesse, Eric Allender, and David A. Mix Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *J. Comput. Syst. Sci.*, 65(4): 695–716, 2002.

[62] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. *Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-34188-5. doi: 10.1007/978-3-642-34188-5_8.

[63] Carolo E. Hindenburg. *In Nitinomii Dignitatum Exponentis Indeterminati*. PhD thesis, University of Göttingen, 1779.

[64] Zheng Huang, Lingli Wang, Yakov Nasikovskiy, and Alan Mishchenko. Fast Boolean matching based on NPN classification. In *Proceedings of the Int'l Conf. on Field-Programmable Technology (FPT)*, pages 310–313, 2013.

[65] Stanley Leonard Hurst, David M. Miller, and Jon C. Muzio. *Spectral Techniques in Digital Logic*. Academic Press, 1985.

[66] John R. Isbell. Median algebra. *Trans. of the Amer. Math. Soc.*, 260(2), 1980.

[67] Valentine Kabanets and Jin-Yi Cai. Circuit minimization problem. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC'00, pages 73–79, New York, NY, USA, 2000. ACM. ISBN 1-58113-184-4. doi: 10.1145/335305.335314.

[68] George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon. Resolution and Parallelizability: Barriers to the Efficient Parallelization of SAT Solvers. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, pages 481–488, 2013.

[69] Donald E. Knuth. *The Art of Computer Programming*, volume 4A. Addison-Wesley, Upper Saddle River, New Jersey, 2011. ISBN 978-0201038040.

[70] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley, Reading, Massachusetts, 2015. ISBN 978-0-13-439760-3.

[71] Arist Kojevnikov, Alexander S. Kulikov, and Grigory Yaroslavtsev. Finding efficient circuits using SAT-solvers. In *Theory and Applications of Satisfiability Testing*, pages 32–44, 2009. ISBN 3642027768.

[72] Kun Kong, Yun Shang, and Rugian Lu. An Optimized Majority Logic Synthesis Methodology for Quantum-Dot Cellular Automata. *IEEE Transactions on Nanotechnology*, 9(2): 170–183, March 2010. ISSN 1536-125X. doi: 10.1109/TNANO.2009.2028609.

[73] Hermann Kopetz. *Internet of Things*, pages 307–323. Springer US, Boston, MA, 2011. ISBN 978-1-4419-8237-7. doi: 10.1007/978-1-4419-8237-7_13. URL https://doi.org/10.1007/978-1-4419-8237-7_13.

[74] Andreas Kuehlmann, Malay K. Ganai, and Viresh Paruthi. Circuit-based Boolean reasoning. In *Proceedings of the Design Automation Conference (DAC)*, 2001. ISBN 1-58113-297-2. doi: 10.1109/DAC.2001.156141.

[75] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 21 (12):1377–1394, 2002.

[76] Alexander S. Kulikov. Improving circuit size upper bounds using sat-solvers. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 305–308, March 2018. doi: 10.23919/DATE.2018.8342026.

[77] Eugene L. Lawler. An approach to multilevel Boolean minimization. *J. ACM*, 11(3): 283–295, 1964.

[78] Daesung Lee, William S. Lee, Chen Chen, Farzan Fallah, John Provine, Soogine Chong, John Watkins, Roger T. Howe, H. S Philip Wong, and Subhasish Mitra. Combinational Logic Design Using Six-Terminal NEM Relays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(5):653–666, 2013. ISSN 02780070. doi: 10.1109/TCAD.2012.2232707.

[79] William C.Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. *Bell Systems Technical Journal*, 38:989–999, 1959. ISSN 00058580. doi: 10.1002/j.1538-7305.1959.tb01585.x.

[80] Craig S. Lent and Paul D. Tougaw. A device architecture for computing with quantum dots. *Proceedings of the IEEE*, 85(4):541–557, April 1997. ISSN 0018-9219. doi: 10.1109/5.573740.

[81] Nan Li and Elena Dubrova. AIG rewriting using 5-input cuts. In *Proceedings of the Int'l Conf. on Computer Design (ICCD)*, pages 429–430, 2011.

[82] Fabrizio Luccio and Linda Pagli. On a New Boolean Function with Applications. *IEEE Transactions on Computers*, 48(3):296–310, 1999. doi: 10.1109/12.754996.

[83] Eugene M. Luks. Isomorphism of Graphs of Bounded Valence Can Be Tested in Polynomial Time. *Journal of Computer and System Sciences*, 25(1):42–65, 1982. ISSN 10902724. doi: 10.1016/0022-0000(82)90009-5.

[84] Valavan Manohararajah, Stephen Dean Brown, and Zvonko G. Vranesic. Heuristics for area minimization in LUT-based FPGA technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 25(11):2331–2340, 2006.

[85] Edward J. McCluskey. Minimization of Boolean Functions. *Bell System Technical Journal*, 35(6):1417–1444, 1956.

[86] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, {II}. *Journal of Symbolic Computation*, 60(0):94 – 112, 2014. ISSN 0747-7171. doi: http://dx.doi.org/10.1016/j.jsc.2013.09.003.

[87] Jin Miao, Andreas Gerstlauer, and Michael Orshansky. Approximate logic synthesis under general error magnitude and frequency constraints. In *Proceedings of the IEEE/ACM Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 779–786, 2013.

[88] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994. ISBN 9780070163331.

[89] Alan Mishchenko. An Approach to Disjoint-Support Decomposition of Logic Functions. Technical report, Portland State University, 2001.

[90] Alan Mishchenko and Robert K. Brayton. Integrating an AIG Package, Simulator, and SAT Solver. In *Proceedings of the Int'l Workshop on Logic Synthesis (IWLS)*, 2018.

[91] Alan Mishchenko and Marek A. Perkowski. Fast heuristic minimization of exclusive-sums-of-products. In *Proc. RM Workshop*, pages 242–250, 2001.

[92] Alan Mishchenko, Satrajit Chatterjee, Roland Jiang, and Robert K. Brayton. FRAIGs: a unifying representation for logic synthesis and verification. Technical report, UC Berkeley, 2005.

[93] Alan Mishchenko, Satrajit Chatterjee, and Robert K. Brayton. DAG-aware AIG rewriting: A Fresh Look at Combinational Logic Synthesis. In *Proceedings of the Design Automation Conference (DAC)*, pages 532–535, 2006.

[94] Alan Mishchenko, Jin S. Zhang, Subarna Sinha, Jerry R. Burch, Robert K. Brayton, and Malgorzata Chrzanowska-Jeske. Using simulation and satisfiability to compute flexibilities in boolean networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(5):743–755, May 2006. ISSN 0278-0070. doi: 10.1109/TCAD.2005.860955.

[95] Alan Mishchenko, Robert K. Brayton, Jie-hong Roland Jiang, and Stephen Jang. SAT-Based Logic Optimization and Resynthesis. In *Proceedings of the Int'l Workshop on Logic Synthesis (IWLS)*, 2007.

[96] Alan Mishchenko, Satrajit Chatterjee, and Robert K. Brayton. Improvements to technology mapping for LUT-based FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 26(2):240–253, 2007.

[97] Alan Mishchenko, Satrajit Chatterjee, and Robert K. Brayton. Improvements to technology mapping for LUT-based FPGAs. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 26, pages 240–253, 2007. ISBN 1595932925. doi: 10.1109/TCAD.2006.887925.

[98] Alan Mishchenko, Sungmin Cho, Satrajit Chatterjee, and Robert K. Brayton. Combinational and Sequential Mapping with Priority Cuts. In *Proceedings of the IEEE/ACM Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 354–361, 2007.

[99] Alan Mishchenko, Robert K. Brayton, Jie-Hong R. Jiang, and Stephen Jang. Scalable Don't-care-based Logic Optimization and Resynthesis. *ACM Trans. Reconfigurable Technol. Syst.*, 4(4):34:1–34:23, December 2011. ISSN 1936-7406. doi: 10.1145/2068716.2068720. URL http://doi.acm.org/10.1145/2068716.2068720.

[100] Alan Mishchenko, Robert K. Brayton, Thierry Besson, Sriram Govindarajan, Harm Arts, and Paul van Besouw. Versatile SAT-based remapping for standard cells. In *Proceedings of the Int'l Workshop on Logic Synthesis (IWLS)*, 2016.

[101] Ernest Morris. *The history and art of change ringing*. Chapman & Hall, 1931.

[102] Cody D. Murray and Ryan Williams. On the (Non) NP-Hardness of Computing Circuit Complexity. *Theory of Computing*, 13(4):1–22, 2017. doi: 10.4086/toc.2017.v013a004.

[103] Ryan O'Donnell. *Analysis of Boolean Functions*. Cambridge University Press, 2014. ISBN 9781107038325.

[104] Peichen Pan and Chih-Chang Lin. A New Retiming-based Technology Mapping for LUT-based FPGAs Algorithm. In *FPGA 98*, pages 35–42, 1998. ISBN 8979197851981.

[105] Wolfgang J. Paul. A $2.5n$-lower bound on the combinational complexity of Boolean functions. *SIAM J. Comput.*, 6(3):427–443, 1977.

[106] Marek A. Perkowski and Malgorzata Chrzanowska-Jeske. An Exact Algorithm to Minimize Mixed-Radix Exclusive Sums of Products for Incompletely Specified Boolean Functions. In *Proc. ISCAS*, pages 1652–1655, 1990. doi: 10.1109/ISCAS.1990.112455.

[107] Ana Petkovska, Mathias Soeken, Giovanni De Micheli, Paolo Ienne, and Alan Mishchenko. Fast Hierarchical NPN Classification. In *Field Programmable Logic and Applications*, pages 1–4, 2016.

[108] Willard V. Quine. The Problem of Simplifying Truth Functions. *The American Mathematical Monthly*, 59(8):521–531, 1952.

[109] Rochit Rajsuman. *System-on-a-Chip: Design and Test*. Artech House, Inc., Norwood, MA, USA, 1st edition, 2000. ISBN 1580531075.

[110] Marc D Riedel. *Cyclic Combinational Circuits*. PhD thesis, 2004.

[111] Heinz Riener, Mathias Soeken, Winston J. Haaswijk, Alan Mishchenko, and Giovanni De Micheli. On-the-fly and DAG-aware: Rewriting Boolean Networks with Exact Synthesis. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019.

[112] John Riordan and Claude E. Shannon. The Number of Two-Terminal Series–Parallel Networks. *Journal of Mathematics and Physics*, 1(4):83–93, 1942.

[113] John P. Roth and Richard M. Karp. Minimization Over Boolean Graphs. *IBM Journal of Research and Development*, 6(2):227–238, 1962.

[114] Sean Safarpour, Andreas Veneris, Gregg Baeckler, and Richard Yuan. Efficient SAT-based Boolean Matching for FPGA Technology Mapping. In *DAC*, pages 466–471, 2006.

[115] Shuichi Sakai, Mitsunori Togasaki, and Koichi Yamazaki. A note on greedy algorithms for the maximum weighted independent set problem. *Discrete Applied Mathematics*, 126(2): 313 – 322, 2003. ISSN 0166-218X. doi: https://doi.org/10.1016/S0166-218X(02)00205-6. URL http://www.sciencedirect.com/science/article/pii/S0166218X02002056.

[116] Salvatore Sanfilippo. Redis. https://redis.io, 2009 (initial release).

[117] Tsutomu Sasao. EXMIN2: A Simplification Algorithm for Exclusive-OR-Sum-of Products Expressions for Multiple-Valued-Input Two-Valued-Output Functions. *IEEE Transactions on CAD*, 12(5):621–632, 1993. ISSN 19374151.

[118] Tsutomu Sasao. An Exact Minimization of AND-EXOR Expressions Using Reduced Covering Functions. In *Proc. of the Synthesis and Simulation Meeting and International Interchange*, pages 374–383, 1993.

[119] Bruno Schmitt, Alan Mishchenko, and Robert K. Brayton. SAT-based Area Recovery in Structural Technology Mapping. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 586–591, Jan 2018. doi: 10.1109/ASPDAC.2018.8297386.

[120] Thomas Schneider, Alexander A. Serga, Britta Leven, Burkard Hillebrands, Robert L. Stamps, and Mikhail P. Kostylev. Realization of spin-wave logic gates. *Applied Physics Letters*, 92(2):022505, 2008. doi: 10.1063/1.2834714.

[121] Claus-Peter Schnorr. The combinational complexity of equivalence. *Theor. Comput. Sci.*, 1(4):289–295, 1976.

[122] Rich Schroeppel. A few mathematical experiments. Talk at Experimental Mathematics Workshop, slides at http://richard.schroeppel.name:8015/expmath04-schroeppel-talk.pdf.

[123] Ellen M. Sentovich, Kanwar J. Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.

[124] Claude E. Shannon. The Synthesis of Two-Terminal Switching Circuits. *Bell System Technical Journal*, 28(1):59–98, 1949. ISSN 15387305. doi: 10.1002/j.1538-7305.1949.tb03624.x.

[125] Carsten Sinz. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In *Principles and Practice of Constraint Programming - CP 2005*, pages 827–831, 2005.

[126] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996. ISBN 053494728X.

[127] Neil J. A. Sloane. The On-Line Encyclopedia of Integer Sequences. URL http://oeis.org. Sequence A000370.

[128] Mathias Soeken, Nabila Abdessaied, and Giovanni De Micheli. Enumeration of reversible functions and its application to circuit complexity. In Simon Devitt and Ivan Lanese, editors, *Reversible Computation*, pages 255–270, Cham, 2016. Springer International Publishing. ISBN 978-3-319-40578-0.

[129] Mathias Soeken, Luca G. Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. Optimizing majority-inverter graphs with functional hashing. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 1030–1035, 2016.

[130] Mathias Soeken, Alan Mishchenko, Ana Petkovska, Baruch Sterin, Paolo Ienne, Robert K. Brayton, and Giovanni De Micheli. Heuristic NPN Classification for Large Functions

Using AIGs and LEXSAT Mathias. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 212–227, 2016.

[131] Mathias Soeken, Luca G. Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. Exact synthesis of majority-inverter graphs and its applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017. ISSN 0278-0070. doi: 10.1109/TCAD.2017.2664059.

[132] Mathias Soeken, Giovanni De Micheli, and Alan Mishchenko. Busy Man's Synthesis: Combinational Delay Optimization With SAT. In *Design Automation and Test in Europe*, 2017.

[133] Mathias Soeken, Winston J. Haaswijk, Eleonora Testa, Alan Mishchenko, Luca G. Amarù, Robert K. Brayton, and Giovanni De Micheli. Practical Exact Synthesis. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 309–314, March 2018. doi: 10.23919/DATE.2018.8342027.

[134] Mathias Soeken, Heinz Riener, Winston J. Haaswijk, and Giovanni De Micheli. The EPFL Logic Synthesis Libraries. *CoRR*, abs/1805.05121, 2018. URL http://arxiv.org/abs/1805.05121.

[135] Mathias Soeken, Eleonora Testa, Alan Mishchenko, and Giovanni De Micheli. Pairs of Majority-Decomposing Functions. *Information Processing Letters*, 139:35–38, 2018. ISSN 00200190. doi: 10.1016/j.ipl.2018.07.004.

[136] Ning Song and Marek A. Perkowski. EXORCISM-MV-2 : Minimization of Exclusive Sum of Products Expressions for Multiple-valued Input incompletely Specified Functions. In *Proc. ISMVL*, pages 132–137, 1993.

[137] Ning Song and Marek A. Perkowski. Minimization of Exclusive Sum of Products Expressions for Multiple-Valued Input. *IEEE Trans. on CAD*, 15(4):385–395, 1996.

[138] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Theory and Applications of Satisfiability Testing*, pages 244–257, 2009.

[139] Ko Stoffelen. Optimizing S-box Implementations for Several Criteria using SAT Solvers. *Lecture Notes in Computer Science*, 9783:140–160, 2016. ISSN 16113349. doi: 10.1007/978-3-662-52993-5_8.

[140] Leon Stok. EDA 3.0: time to refactor logic synthesis. In *EPFL Workshop on Logic Synthesis & Verification*, 2015.

[141] Eleonora Testa, Mathias Soeken, Odysseas Zografos, Francky Catthoor, and Giovanni De Micheli. Exact synthesis for logic synthesis applications with complex constraints. 2017.

[142] Hervé J. Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*, pages 130–133, Nov 1990. doi: 10.1109/ICCAD.1990.129860.

[143] Paul D. Tougaw and Craig S. Lent. Logic devices implemented using quantum cellular automata. *Journal of Applied Physics*, 75(3), 1994.

[144] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting Cheng. *Electronic Design Automation: Synthesis, Verification, and Test*. 2009. ISBN 9780123743640. doi: 10.1088/1751-8113/44/8/085201.

[145] Felix Wortmann and Kristina Flüchter. Internet of things. *Business & Information Systems Engineering*, 57(3):221–224, 06 2015.

[146] Sergey Yablonski. The algorithmic difficulties of synthesizing minimal switching circuits. *Problemy Kibernetiki*, 2(1):75–121, 1959.

[147] Wenlong Yang, Lingli Wang, and Alan Mishchenko. Lazy Man's Logic Synthesis. In *Proceedings of the Int'l Workshop on Logic Synthesis (IWLS)*, 2012.

[148] Cunxi Yu, Houping Xiao, and Giovanni De Micheli. Developing Synthesis Flows Without Human Knowledge. In *Proceedings of the Design Automation Conference (DAC)*, pages 50:1–50:6, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5700-5. doi: 10.1145/3195970.3196026. URL http://doi.acm.org/10.1145/3195970.3196026.

[149] Billie J. Zirger and Janet L. Hartley. The effect of acceleration techniques on product development time. *IEEE Transactions on Engineering Management*, 43(2):143–152, May 1996. ISSN 0018-9391. doi: 10.1109/17.509980.

Winston Jason Haaswijk

Personal Information

Winston J. Haaswijk is a PhD student in the Integrated Systems Laboratory (LSI) at EPFL, Lausanne. He received his Bachelor degree in Computer Science from the University of Amsterdam, and his MPhil in Computer Science from the University of Cambridge. His research interests include Boolean satisfiability, exploring novel logic primitives, SAT based synthesis methods, machine learning in general, and applications of machine learning to EDA in particular. He maintains percy, which is a C++ header-only SAT-based exact synthesis library, and one of the EPFL logic synthesis libraries.

Contact Information

Address:	Rue du Valentin 30	Mobile:	(+41) 79 209 65 44
	1004-CH Lausanne	E-mail:	winston.haaswijk@gmail.com
	Switzerland	Web:	whaaswijk.github.io
Citizenship:	The Netherlands		

Educational Background

Sept 2015 - Today
Doctoral Candidate, Computer Science
EPFL, Lausanne, Switzerland
Integrated System Lab (LSI)

Oct 2013 - Jul 2014
MPhil, Advanced Computer Science
University of Cambridge, Cambridge, United Kingdom
The Computer Laboratory

Sep 2008 - Aug 2012
BSc, Computer Science
University of Amsterdam, The Netherlands
Faculty of Science
(Graduated Cum Laude and Cum Honore)

Sep 2001 - Jun 2007
High School Diploma, VWO
Fons VitæLyceum, Amsterdam, The Netherlands

Professional Experience

April 2013 - June 2014 **University of Cambridge**, Cambridge, United Kingdom
MPhil Dissertation - Program Synthesis in HOL

In this project I showed how the HOL theorem prover can be used to construct provably correct programs from specifications in an interactive way. I also showed how the resulting programs can compiled to binary executables in a verifiably correct way, thus creating a verifying pipeline from formal specifications to executable programs.

June 2017 - Sep 2017 **Synopsys Inc.**, Mountain View, California, USA
Technical Intern

At Synopsys, I worked in the Design Group, which is responsible for projects in the fields of logic synthesis, physical synthesis, placement, and RC estimation. While at Synopsys, I worked under the super the supervision of Dr. Luca Amarù and Dr. Jiong Luo. Specifically, my project focused on the optimization of XOR-heavy logic. Partial results were later published at ICECS'18.

Sep 2010 - Oct 2013 **SecuReceipt B.V.**, Amsterdam, The Netherlands
Lead Developer

SecuReceipt is an ambitious startup company that offers a modern and easy expense management solution. I was involved from the start, and helped lead its technical development. SecuReceipt develops new technologies that are now in use by numerous companies. I aided in the development of the web platform, and led the development of the mobile platform.

Feb 2011 - Aug 2012 **University of Amsterdam**, Amsterdam, The Netherlands
Bachelor Thesis Project - Robust applications in the open internet

In this project I examined the advent of Cloud Computing and how its flexibility might be used to create robust applications. I presented several robustness techniques, and proposed an application architecture based on SOA to design robust applications on the best-effort infrastructure of the Cloud.

Sep 2009 - Jan 2010 **University of Amsterdam**, Amsterdam, The Netherlands
Student mentor

I was responsible for helping new Computer Science students make their way in the university. They could come to me with any problems or questions.

Mar 2008 - Sep 2010 **Hippo B.V.**, Amsterdam, The Netherlands
Junior Web Developer

Hippo is a company that develops an award-winning open source CMS. At Hippo I was part of the project team, implementing web-based solutions for government and large companies. The technologies we used included Java, Servlets, JSP, and the Hippo CMS itself.

Sep 2007 - Mar 2008 **Haaswijk Software Engineering**, Amsterdam, The Netherlands
Freelance Web Developer

I worked on several projects as a freelance web developer. These included implementing content management systems and web sites for small companies. Most of this work was done by using the LAMP stack (Linux, Apache, MySQL, and PHP).

Technical Skills

Programming Languages	*(Advanced knowledge)* **C/C++, Python,** **Shell Scripting,** **Javascript** **HTML+CSS** **Objective-C, C#, Java,** **Servlets, JSP, SQL,** **PHP, JSON, XML** *(Beginner knowledge)* **Haskell,** **Assembly Language,** **ML (OCaml and Standard ML)**
Version Control Systems	**GIT, SVN, Mercurial**
Logic Synthesis Development	(academic experience, industry experience)
Algorithms and Data Structures	(academic experience, teaching experience)
Theory of Computation	(academic experience, teaching experience)
Operating Systems, Compilers	(academic experience)
Mobile Application Development	(industry experience)
Web Development	(industry experience)
System Administration	(industry experience)

Languages

Dutch	Native	
English	Fluent	(TOEFL iBT score 116/120)
German	Beginner	
French	Beginner	

Interests and Activities

Logic Synthesis, Boolean Satisfiability, Program Synthesis, Automated Reasoning, Machine Learning, Formal Verification, Circuit Complexity, Automated Theorem Proving, Entrepreneurship

Software. I maintain the *percy* exact synthesis package, which can be found at https://github.com/whaaswijk/percy. *percy* is an advanced, header-only, SAT-based exact synthesis package for the synthesis of optimum multi-level logic networks.

Teaching. I have taught advanced courses on *Theory of Computation* and *Electronic Design Automation*

Hobbies: Creative Writing, Traveling, Hiking, Philosophy, Politics, Football, Cinema, Fitness

Publications

[1] Zhufei Chu, Winston J. Haaswijk, Mathias Soeken, Lunyao Wang, Yinshui Xia, and Giovanni De Micheli. "Exact Synthesis of Boolean Functions in Majority-of-five Form". In: *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*. Sapporo, Hokkaido, Japan, May 2019.

[2] Winston J. Haaswijk, Mathias Soeken, Alan Mishchenko, and Giovanni De Micheli. "SAT-Based Exact Synthesis: Encodings, Topology Families, and Parallelism". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2019).

[3] Heinz Riener, Mathias Soeken, Winston J. Haaswijk, Alan Mishchenko, and Giovanni De Micheli. "On-the-fly and DAG-aware: Rewriting Boolean Networks with Exact Synthesis". In: *Proceedings of Design, Automation and Test in Europe (DATE)*. Florence, Italy, Mar. 2019.

[4] Heinz Riener, Eleonora Testa, Winston J. Haaswijk, Alan Mishchenko, Luca G. Amarù, Giovanni De Micheli, and Mathias Soeken. "Scalable Generic Logic Synthesis: One Approach to Rule Them All". In: *Proceedings of the Design Automation Conference (DAC)*. Las Vegas, Nevada, USA, June 2019.

[5] Winston J. Haaswijk, Luca G. Amarù, Patrick Vuillod, Jiong Luo, Mathias Soeken, and Giovanni De Micheli. "Integrated ESOP Refactoring for Industrial Designs". In: *Proceedings of IEEE International Conference on Electronics, Circuits, and Systems (ICESC)*. Bordeaux, France, Dec. 2018.

[6] Winston J. Haaswijk, Edo Collins, Benoit Seguin, Mathias Soeken, Sabine Süsstrunk, Frédéric Kaplan, and Giovanni De Micheli. "Deep Learning for Logic Synthesis Algorithms". In: *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*. Florence, Italy, May 2018.

[7] Winston J. Haaswijk, Alan Mishchenko, Mathias Soeken, and Giovanni De Micheli. "SAT Based Exact Synthesis Using DAG Topology Families". In: *Proceedings of the Design Automation Conference (DAC)*. DAC'18. San Francisco, California, June 2018, 53:1–53:6. ISBN: 978-1-4503-5700-5. DOI: 10.1145/3195970.3196111. URL: http://doi.acm.org/10.1145/3195970.3196111.

[8] Mathias Soeken, Winston J. Haaswijk, Eleonora Testa, Alan Mishchenko, Luca G. Amarù, Robert K. Brayton, and Giovanni De Micheli. "Practical Exact Synthesis". In: *Proceedings of Design, Automation and Test in Europe (DATE)*. Dresden, Germany, Mar. 2018, pp. 309–314. DOI: 10.23919/DATE.2018.8342027.

[9] Mathias Soeken, Heinz Riener, Winston J. Haaswijk, and Giovanni De Micheli. "The EPFL Logic Synthesis Libraries". In: *CoRR* abs/1805.05121 (2018). arXiv: 1805.05121. URL: http://arxiv.org/abs/1805.05121.

[10] Eleonora Testa, Mathias Soeken, Luca G. Amarù, Winston J. Haaswijk, and Giovanni De Micheli. "Mapping Monotone Boolean Functions into Majority". In: *IEEE Transactions on Computers* (2018). DOI: 10.1109/TC.2018.2881245. URL: http://infoscience.epfl.ch/record/261167.

[11] Luca G. Amarù, Mathias Soeken, Winston J. Haaswijk, Eleonora Testa, Patrick Vuillod, Jiong Luo, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. "Multi-level Logic Benchmarks: An Exactness Study". In: *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*. Chiba, Chiba Prefecture, Japan, Jan. 2017.

[12] Winston J. Haaswijk, Edo Collins, Benoit Seguin, Mathias Soeken, Sabine Süsstrunk, Frédéric Kaplan, and Giovanni De Micheli. "Deep Learning for Logic Optimization". In: *Proceedings of the International Workshop on Logic Synthesis (IWLS)*. Austin, Texas, USA, June 2017.

[13] Winston J. Haaswijk, Mathias Soeken, Luca G. Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. "A Novel Basis for Logic Rewriting". In: *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*. Chiba, Chiba Prefecture, Japan, Jan. 2017.

[14] Winston J. Haaswijk, Eleonora Testa, Mathias Soeken, and Giovanni De Micheli. "Classifying Functions with Exact Synthesis". In: *Proceedings of the International Symposium on Multiple-Valued Logic (ISMVL)*. Novi Sad, Serbia, May 2017.

[15] Winston J. Haaswijk, Mathias Soeken, Luca G. Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. "LUT Mapping and Optimization for Majority-Inverter Graphs". In: *Proceedings of the International Workshop on Logic Synthesis (IWLS)*. Austin, Texas, USA, June 2016.

[16] Winston J. Haaswijk, Luca G. Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. "NEM Relay Design with Biconditional Binary Decision Diagrams". In: *Proceedings of IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*. Boston, Massachusetts, USA, July 2015.

**Ce document a été imprimé au Centre d'impression EPFL,
imprimerie climatiquement neutre, certifiée myClimate.**