Exploiting Satisfiability Solvers for Efficient Logic Synthesis

THÈSE N^{O} 7866 (2017)

PRÉSENTÉE LE 25 SEPTEMBRE 2017
À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'ARCHITECTURE DES PROCESSEURS
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Ana PETKOVSKA

acceptée sur proposition du jury:

Prof. V. Kunčak, président du jury
Prof. P. Ienne, directeur de thèse
Dr A. Mishchenko, rapporteur
Prof. I. Markov, rapporteur
Prof. G. De Micheli, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2017

Дом е таму каде што боли.
Дом е таму каде што ви се вестите.
Таму каде што ве сакаат и таму каде што ве мразат.
— Горан Стефановски
Интервју, проект Дванаесет

Home is where it hurts.
Home is where your news are.
Where they love you and where they hate you.
— Goran Stefanovski
Interview, project Twelve

To Vase and my parents.

Abstract

Logic synthesis is an important part of *electronic design automation (EDA)* flows, which enable the implementation of digital systems. As the design size and complexity increase, the data structures and algorithms for logic synthesis must adapt and improve in order to keep pace and to maintain acceptable runtime and high-quality results. Large circuits were often represented using *binary decision diagrams (BDDs)* that were rapidly adopted by logic synthesis tools beginning in the 1980s. Nowadays, BDD-based algorithms are still enhanced, but the possibilities for improvement are somewhat saturated after some 35 years of research.

Alternatively, the first EDA applications that exploit *Boolean satisfiability (SAT)* were developed in the 1990s. Despite the worst-case exponential runtime of SAT solvers, rapid progress in their performance enabled the creation of efficient SAT-based algorithms. Yet, logic synthesis started using SAT solvers more diffusely only in the last decade. Therefore, thorough research is still required both for exploiting SAT solvers and for encoding logic synthesis problems into SAT. Our main goal in this thesis is to facilitate and promote the further integration of SAT solvers into EDA by proposing and evaluating novel SAT-based algorithms that can be used as building blocks in logic synthesis tools.

First, we propose a rapid algorithm for LEXSAT, which generates satisfying assignments in lexicographic order. We show that LEXSAT can bring canonicity, which guarantees the generation of unique results, when using SAT solvers in EDA applications. Next, we present a new SAT-based algorithm that progressively generates irredundant *sums of products (SOPs)*, which still play a crucial role in many logic synthesis tools. Using LEXSAT, for the first time, we can generate canonical SAT-based SOPs that, much like BDD-based SOPs, are unique for a given function and variable order but could relax canonicity in order to improve speed and scalability. Unlike BDDs, due to its progressive nature, our algorithm can generate partial SOPs for applications that can work with incomplete circuit functionality. It is noteworthy that both LEXSAT and the SAT-based SOPs are applicable beyond logic synthesis and EDA. Finally, we focus on *resubstitution*, which reimplements a given Boolean function as a new function that depends on a set of existing functions called divisors. We propose the *carving interpolation* algorithm that, unlike the traditional Craig interpolation, forces the use of a specific divisor as an input of the new function. This is particularly useful for global circuit restructuring and for some synthesis-based *engineering change order (ECO)* algorithms. Furthermore, we compare two existing SAT-based methodologies for resubstitution, which are used for post-mapping logic optimisation. The first methodology combines SAT-based functional dependency checking and Craig interpolation that are also used for our carving

interpolation; the second methodology is based on cube enumeration and is similar to the SAT-based SOP generation.

The initial implementations of our novel SAT-based algorithms offer either better performance or new features, or both, compared to their state-of-the-art versions. As the results indicate, a further thorough development of SAT-based algorithms for logic synthesis, like the one performed for BDDs in the past, can help overcome existing limitations and keep up with growing designs and design complexity.

Keywords: electronic design automation, logic synthesis, Boolean satisfiability, SAT solvers.

Résumé

La synthèse logique est une partie importante des flots de *conception assistée par ordinateur pour l'électronique* (*CAO électronique*; *electronic design automation, EDA*) permettant l'implémentation des systèmes digitaux. L'augmentation en taille et en complexité des designs nécessite l'adaptation des structures de données et des algorithmes utilisés dans la synthèse logique afin de rester compétitif et de maintenir un temps d'exécution acceptable et des résultats de haute-qualité. Les grands circuits étaient souvent représentés par des *diagrammes de décision binaire* (*binary decision diagrams, BDDs*) qui ont été rapidement adoptés par les outils de synthèse logique au début des années 1980. De nos jours les algorithmes basés sur les diagrammes de décision binaire sont toujours en développement, cependant les possibilités d'amélioration se sont quelque peu saturées après environ 35 ans de recherche.

De manière alternative, les premières applications de CAO électronique exploitant la *satisfaisabilité booléenne* (*Boolean satisfiability, SAT*) ont été développées dans les années 1990. Malgré le temps d'exécution exponentiel dans le pire cas des solveurs SAT, la progression rapide de leur performance a permis de créer des algorithmes efficaces qui sont basés sur SAT. Cependant, la synthèse logique a commencé à utiliser les solveurs SAT plus couramment seulement dans la dernière décennie. C'est pourquoi il est nécessaire d'approfondir encore la recherche pour trouver des meilleures façons d'exploiter des solveurs SAT ainsi que d'encoder des problèmes de synthèse logique en SAT. Notre but principal dans cette thèse est de faciliter et de promouvoir l'intégration en cours des solveurs SAT dans la CAO électronique en proposant et en évaluant des algorithmes novateurs basés sur SAT qui peuvent être utilisés en tant que blocs de construction dans les outils de synthèse logique.

En premier, nous proposons un algorithme rapide pour LEXSAT qui génère des affectations satisfaisantes en ordre lexicographique. Nous montrons que LEXSAT peut apporter de la canonicité, garantie de génération de résultats uniques, lorsqu'on utilise des solveurs SAT dans les applications de CAO électronique. Ensuite, nous présentons un nouvel algorithme basé sur SAT générant de façon progressive des *sommes de produits* (*sums of products, SOP*) non redondants qui jouent toujours un rôle crucial dans nombres d'algorithmes de synthèse logique. Avec LEXSAT, nous pouvons pour la première fois générer des SOPs canoniques basées sur SAT qui, tout comme les SOPs basées sur des diagrammes de décision binaire, sont uniques pour une fonction et un ordonnancement des variables, mais peuvent assouplir la canonicité afin d'améliorer la vitesse et la scalabilité. Contrairement aux diagrammes de décision binaire, notre algorithme peut, de par sa nature progressive, aussi générer les SOPs partiels pour des applications qui peuvent travailler avec la fonctionnalité incomplète d'un circuit. Il est

remarquable que LEXAT et les SOPs basées sur SAT sont applicables au-delà de la synthèse logique et de la CAO électronique. Finalement, nous nous concentrons sur la *resubstitution*, qui ré-implémente une fonction booléenne donnée comme une nouvelle fonction dépendant sur un ensemble de fonctions existantes, appelées diviseurs. Nous proposons l'algorithme d'*interpolation à ciselage* (*carving interpolation*) qui, contrairement à l'interpolation traditionnelle de Craig, impose l'utilisation d'un diviseur spécifique en tant qu'entrée de la nouvelle fonction. Ceci est particulièrement utile pour la restructuration globale de circuit ainsi que pour quelques algorithmes d'*engineering change order (ECO)* basés sur la synthèse logique. De plus, nous comparons deux méthodologies existantes pour la resubstitution basés sur SAT, utilisées pour l'optimisation logique après le *technology mapping*. La première méthodologie combine la vérification de dépendances fonctionnelles basées sur SAT et l'interpolation de Craig, également utilisées dans notre interpolation à ciselage ; la deuxième méthodologie est basée sur l'énumération de cubes et est similaire à la génération de SOPs basée sur SAT.

Les implémentations initiales de nos algorithmes novateurs basés sur SAT offrent soit de meilleures performances soit de nouvelles fonctionnalités, ou toutes les deux, en comparaison à leurs versions de pointe. Comme les résultats l'indiquent, un développement plus approfondi des algorithmes basés sur SAT pour la synthèse logique, comme celui effectué pour les diagrammes de décision binaire dans le passé, peut aider à surmonter les limitations existantes et à rester en phase avec la croissance et la complexité des designs.

Mots clefs : conception assisté par ordinateur pour l'électronique, synthèse logique, satisfaisabilité booléenne, solveurs SAT.

Резиме

Логичката синтеза (*logic synthesis*) е значаен дел од системите за автоматизација на електронски дизајн (*electronic design automation, EDA*), кои овозможуваат изработка на дигитални системи. Како што големината и комплексноста на дизајните расте, структурите на податоци и алгоритмите за логичка синтеза мора да се прилагодуваат и усовршуваат за да останат конкурентни и за да одржат прифатливо време на извршување и резултати со висок квалитет. Големите дигитални кола беа често прикажувани со бинарни дијаграми за одлука (*binary decision diagrams, BDDs*) кои беа брзо прифатени од алатките за логичка синтеза почнувајќи од 1980-тите години. Денес, *BDD*-базираните алгоритми сè уште се унапредуваат, но можностите за подобрување се донекаде заситени после околу 35 години истражување.

Алтернативно, првите *EDA* апликации кои користат Булова задоволителност (*Boolean satisfiability, SAT*) беа развиени во 1990-тите години. И покрај експоненцијалното најлошо време на извршување на *SAT* решавачите (*SAT solvers*), брзиот напредок во нивните перформанси овозможи создавање на ефикасни *SAT*-базирани алгоритми. Сепак, SAT решавачите почнале да се користат пораспространето во логичката синтеза дури во последната деценија. Поради тоа, сè уште има потреба од темелно истражување за искористување на *SAT* решавачите, како и за претставување на проблемите од логичка синтеза како *SAT* проблеми. Нашата главна цел во оваа дисертација е да ја олесниме и промовираме понатамошната интеграција на *SAT* решавачите во *EDA*, со тоа што предлагаме и евалуираме нови *SAT*-базирани алгоритми кои може да се користат како основни елементи во алатките за логичка синтеза.

Прво, ние предложуваме брз алгоритам за *LEXSAT*, кој генерира задоволувачки вредности (*satisfying assignments*) по лексикографски редослед. Покажуваме дека *LEXSAT* може да овозможи каноничност (*canonicity*), која гарантира создавање на уникатни резултати, при користење на *SAT* решавачи во *EDA* апликации. Следно, претставуваме нов *SAT*-базиран алгоритам кој постепено генерира сума од производи (*sum of products, SOP*) без редундантност, кои сè уште имаат клучна улога во многу алатки за логичка синтеза. Со користење на *LEXSAT*, за прв пат, можеме да генерираме канонски *SAT*-базирани *SOP*-а кои, слично на *BDD*-базираните *SOP*-а, се единствени за дадена функција и редослед на променливите, но со можност да се релаксира каноничноста за да се подобри бр-

зината и приспособливоста. За разлика од *BDD*-ата, поради постепената природа, нашиот алгоритам може да генерира парцијални *SOP*-а за апликации кои можат да работат со нецелосна функционалност на колата. Значајно е што и *LEXSAT* и *SAT*-базираните *SOP*-а се применливи пошироко од логичката синтеза и *EDA*. На крај, се фокусираме на ресупституција (*resubstitution*), која реимплементира дадена Булова функција како нова функција која зависи од множество од постоечки функции наречен делители. Ние го предложуваме алгоритамот за интерполација со отсекување (*carving interpolation*) која за разлика од традиционалната Крегова интерполација (*Craig interpolation*) ја принудува употребата на одреден делител како влез на новата функција. Ова е особено корисно за глобално реструктуирање на кола и за некои алгоритми базирани на *engineering change order (ECO)* алгоритми. Покрај тоа, споредуваме две постоечки *SAT*-базирани методологии за ресупституција кои се користат при логичка оптимизација после технолошко мапирање (*technology mapping*). Првата методологија соединува *SAT*-базирана проверка за функционална зависност и Крегова интерполација кои исто така се употребуваат за нашата интерполација со отсекување; втората методологија се базира на набројување на коцки (*cube enumeration*) и е слична со создавањето на SAT-базирани *SOP*-а.

Првобитните имплементации на нашите нови *SAT*-базирани алгоритми нудат или подобри перформанси или нови функции или и двете, во споредба со нивните современи верзии. Како што резултатите укажуваат, понатамошен темелен развој на *SAT*-базиранте алгоритми за логичка синтеза, како оној кој беше извршен за *BDD*-ата во минатото, може да помогне да се надминат постоечките ограничувања и да се остане во чекор со растот на дизајните и нивната комплексност.

Клучни зборови: автоматизација на електронски дизајн, логичка синтеза, Булова задоволителност, *SAT* решавачи.

Acknowledgements

Obtaining this thesis was made possible and enjoyable by many people that have left their footprints in my life.

I am deeply grateful to my advisor, Paolo Ienne, first, for encouraging me to start the PhD studies and, second, for guiding me through the process. He gave me freedom and vast support in pursuing projects for which I was passionate. His advices and constructive criticism empowered me to overcome many obstacles. I have learned a lot from him on both professional and personal level as he had a role of both a mentor and a friend.

Although the logic synthesis community is quite small compared to others, I had the privilege of working closely with and learning from some of the top people in the field.

First and foremost, I am extremely grateful to Alan Mishchenko for sharing with us many research opportunities and being always available for discussions both over the Internet and in person. He provided me with tremendous support, inspiring advices, and profound belief in my abilities throughout the course of my PhD studies. I would also like to extend my gratitude to Mathias Soeken who has kindly answered many questions and helped me to verify many ideas I had in the last years. I am also thankful to him for his help, valuable comments and discussions while I was writing this thesis. I would like to thank David Novo for his unwavering guidance and support in the first part of my PhD studies. I also wish to thank Ajay K. Verma for introducing me to logic synthesis during my internship at EPFL in 2009. I have learned a lot from all of them, and they all had a major impact on my life as collaborators, colleagues, (un)official mentors, and friends.

I would also like to express my gratitude to Robert K. Brayton, Luca Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli with whom I had the pleasure to collaborate on various publications, some of which are used in this thesis. I also had great pleasure of working on challenging logic synthesis projects with some brilliant students, including Grégoire Hirt, Mertcan Temel, Polina Zablotskaia, and Sun Chenfan.

I would like to especially thank the members of my thesis committee Alan Mishchenko, Igor Markov, Giovanni De Micheli, and Viktor Kunčak for the time that they dedicated to this task, as well as for their constructive feedback and for the insightful discussions we had.

I would also like to thank Sahand Kashani-Akhavan and Chantal Schneeberger for providing and reviewing the French version of the abstract. Likewise, I would like to thank Andrew Becker and Biljana Vörös for proofreading parts of this thesis.

Acknowledgements

The time spent at Processor Architecture Laboratory (LAP) has been so valuable because of the people comprising it. I was fortunate to interact and spend amazing time with some former and present members of LAP, including Ajay, Ali, Andrew, Chantal, David, Grace, Hadi, Lana, Madhura, Mikhail, Mirjana, Nithin, Paolo, René, Sahand, Thomas, and Xavier. I would like to express my deep gratitude to Chantal Schneeberger, Grace Zgheib, and Nithin George for their friendship and constant support from the beginning of my PhD; but especially to Grace Zgheib with whom I was lucky to share the office and work with, and who became a close friend on whom I could always rely.

The Macedonian crew—including Ana, Biljana, Darko, Dino, Emil, Filip, Gorica, Igor, Jovanche, Marjan, Paulina, Slavica, Stefan, and many others—made the stay in Switzerland fun and less homesick. I am especially grateful to Gorica for being always there for me. Since the beginning of my PhD studies, I have shared with her joyful and hard moments, both from my personal and professional life.

Finally, my deepest gratitude goes to my family. The unconditional love and selfless support of my parents, Vesna Stefanovska and Borcho Stefanovski, made me the person I am today and enabled me to achieve many goals in life, including finishing these PhD studies. Their passion for learning, teaching, and new technologies, which they have been transferring to me since the day I was born, have lead me where I am now. I would like to thank them for everything they have done for me. I am thankful to my sister, Eva Stefanovska, for her love and support, and for being there for me whenever I need her. Last but not least, I will be eternally thankful to my husband, Vase Petkovski, who agreed to undertake this journey with me. His love, devotion, and support bring warmth and vividness every day of my life; whereas his patience, care and encouragement give me strength and make it easier for me to go through hard periods and overcome many challenges while actualising my aspirations.

Lausanne, July 17, 2017 Ana Petkovska

Contents

List of abbreviations

#SAT	counting satisfiability
AIG	and-inverter graph
AllSAT	all solutions satisfiability
ASIC	application specific integrated circuit
BBDD	biconditional binary decision diagram
BDD	binary decision diagram
CNF	conjunctive normal form
DAG	directed acyclic graph
DLN	dependency logic network
DNF	disjunctive normal form
ECO	engineering change order
EDA	electronic design automation
FPGA	field-programmable gate array
ISOP	irredundant sum of products
LEX-UNSAT	lexicographic unsatisfiability
LEXSAT	lexicographic satisfiability
LUT	look-up table
MAX-SAT	maximum satisfiability
MIG	majority-inverter graph
NPN	negation-permutation-negation
PI	primary input
PLA	programmable logic array
PO	primary output
QBF	quantified Boolean formula
ROBDD	reduced ordered binary decision diagram

SAT	Boolean satisfiability
SMT	satisfiability modulo theories
SOP	sum of products
SPFD	sets of pairs of functions to be distinguished
TBDD	transformation binary decision diagram
TFI	transitive fanin
TFO	transitive fanout
UNSAT	unsatisfiable
VLIW	very long instruction word
ZDD	zero-suppressed decision diagram

List of Figures

List of Tables

1 Satisfiability Solvers for Electronic Design Automation

The design complexity is growing, EDA should keep up the pace

Electronic design automation (EDA) tools ease the implementation of digital systems by providing an automated process for translating a high-level description of a design into its final chip implementation. Research and development in EDA began in the 1960s, soon after the development of the integrated circuit [Wang et al., 2009]. Later, an extensive academic research was initiated and accelerated the advancement. As a consequence, the EDA tools have followed the continuous growth in design size and can implement digital systems with more than a billion transistors. This growth in design size and complexity is mainly attributable to advancements in hardware. The revised Moore's law predicts that transistor density doubles approximately every two years [Moore, 1975]. Despite the fear that this rate of growth might be impossible to maintain any further, the prediction has remained largely accurate. Many experts, including Moore himself [Courtland, 2015], are convinced that this can be continued at least through the following decade. Moreover, more complex technologies such as *field-programmable gate arrays (FPGAs)* and 3D integrated circuits are increasingly prevalent in the market. Today, we have a commercially available single-chip processor with about 7.2 billion transistors [Alcorn, 2016] and an FPGA with about 30 billion transistors implementing 5.5 million logic elements [Altera, 2016; Rubenstein, 2016]. In order to use them efficiently, EDA tools have to be upgraded with scalable techniques to keep up with the size and complexity of modern designs, while maintaining the tools' performance and ideally achieving an improved quality of the final design implementations. This would enable the EDA tools to further accomplish their goal of producing high-quality complex

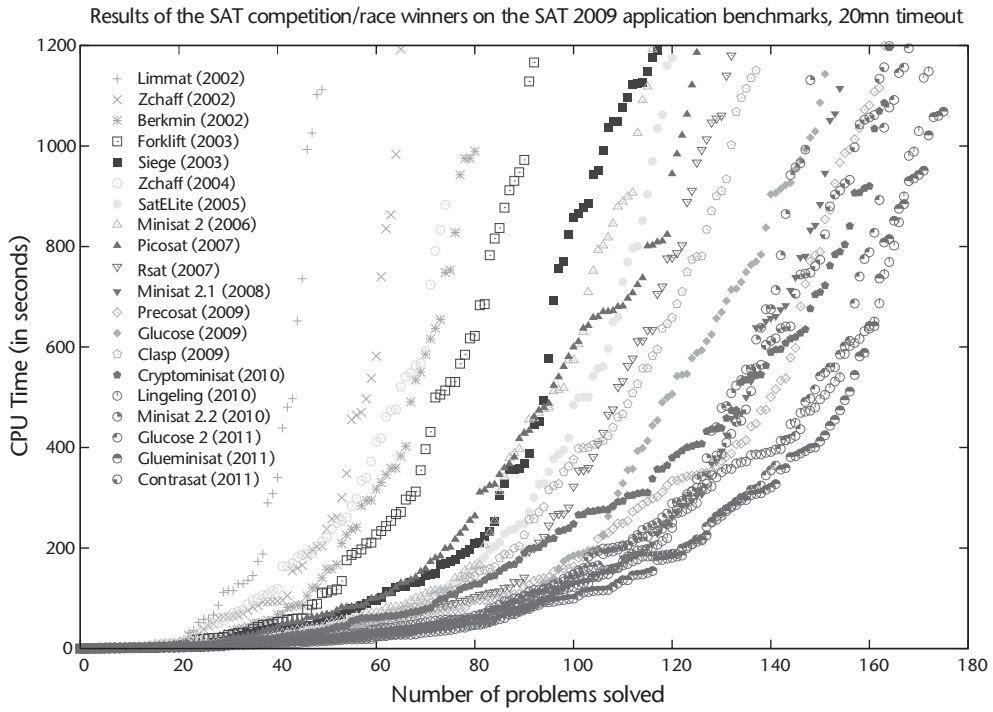


Figure 1.1 – Evolution of the best SAT solvers from 2002 to 2011. The performance of the given solvers is compared using benchmarks from the SAT 2009 competition. For each solver, the cumulative number of solved problems (x-axis) within a specific amount of time (y-axis) is given [Järvisalo et al., 2012].

circuits without design defects in relatively short design time, which allows meeting time-to-market and cost requirements [De Micheli, 1994].

The performance of SAT solvers is continuously improving

On a different note, the *Boolean satisfiability (SAT)* problem, which determines if there exist a variable assignment for which a given propositional formula evaluates to true, received great attention from both a theoretical and a practical point of view. Theoretically, it is unlikely to find an algorithm for solving the SAT problem with a polynomial worst-case time complexity as it is an NP-complete problem [Cook, 1971]. Although the NP-completeness currently leaves us with only deterministic algorithms with worst-case exponential runtime, in practice many modern SAT algorithms, which are implemented in *SAT solvers*, can solve complex instances of real world problems in a reasonable time. We have witnessed a tremendous improvement of the performance of state-of-the-art SAT solvers in the last two decades—SAT problems that could not have been solved in more than an hour are now solvable in

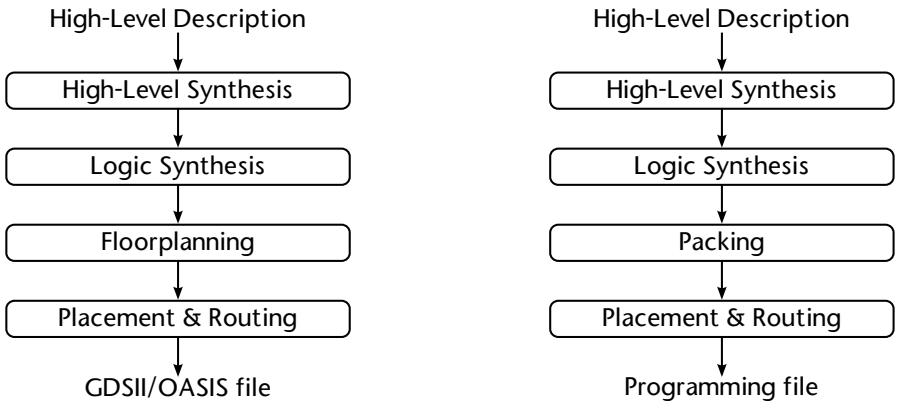
less than a second [Marques-Silva, 2007]. For example, Figure 1.1 shows the evolution of the best solvers from 2002 to 2011 [Järvisalo et al., 2012]. This continuous advancement of SAT solvers can be tracked through the SAT competitions that have started in 1992 and have been organised annually since 2002 [Balyo et al., 2017; Järvisalo et al., 2012]. Their main goal is to stimulate the innovation and development of techniques for SAT solving. Considering the success of and interest in these competitions, it is expected that this trend of improving SAT solvers will continue in the coming years. Another reason for improving SAT solvers is that SAT has already a wide variety of practical applications across a number of domains [Gu et al., 1996; Marques-Silva, 2008], such as artificial intelligence, bioinformatics, software verification, and automated theorem proving, as well as EDA [Claessen et al., 2009; Marques-Silva and Sakallah, 2000].

In the following sections of this chapter, we first present a short history on the evolution of the algorithms for logic synthesis, which is the research subject of this thesis. Next, we provide algorithms from logic synthesis and other EDA stages as examples that already use SAT solvers as the core machinery. Then, we discuss the main challenges of exploiting SAT solvers in logic synthesis. Finally, we summarise our contributions and present the thesis organisation.

1.1 From Truth Tables to SAT Solvers: The Evolution of Logic Synthesis Algorithms

The vital role of logic synthesis as part of EDA flows

EDA flows differ depending on the target technology for implementing the given design. For example, Figure 1.2 shows side-by-side the typical stages of the EDA flows for *application specific integrated circuits (ASICs)* [Smith, 1997] and FPGAs [Betz et al., 1999]. Regardless of the target technology, logic synthesis is part of any EDA flow and has a vital role because its operation has a major impact on the performance of the final implementation. It receives as input a *gate-level netlist* that is composed of basic Boolean logic gates, such as AND, OR, and XOR gates, and inverters. Its main goal is to convert the received netlist into a high-quality mapped netlist of technology blocks without modifying the functionality. This is achieved with the two main steps of logic synthesis. First, it performs technology-independent *logic optimisation* of the gate-level description: it reduces the area and delay [Brayton

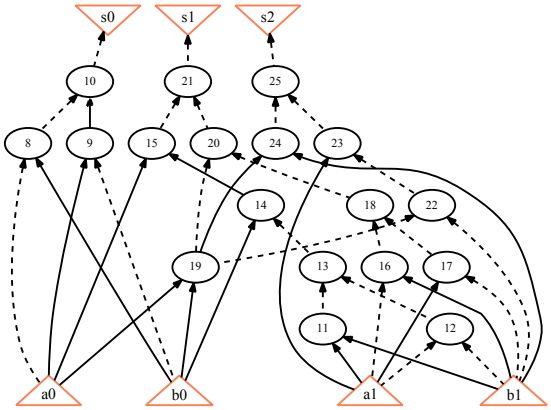


(a) ASIC design flow. **(b)** FPGA design flow.

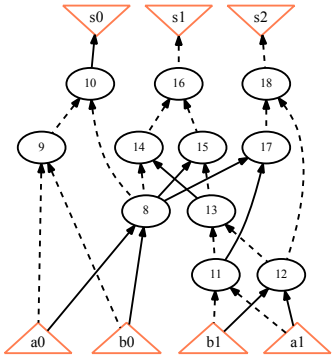
Figure 1.2 – Design flows for ASICs and FPGAs. Although some of the stages are different, logic synthesis is always required. It receives a gate-level netlist from the high-level synthesis stage and outputs a netlist of technology blocks for the floorplanning or packing stage for ASICs and FPGAs, respectively.

et al., 1990]. For a gate-level netlist, its *area* represents the size of the netlist in terms of number of logic gates used to implement it, and its *delay* represents the number of logic levels on the critical path. Once an implementation with a good gate-level area and delay is produced, *technology mapping* algorithms transform the gate-level netlist into a netlist of larger logic blocks whose structure depends on the target technology [De Micheli, 1991]. For example, for ASICs, the logic gates are mapped into a netlist of ASIC blocks that usually represent *standard cells*, which are predesigned logic components that implement a Boolean logic function or a memory element. Whereas, for FPGAs, it creates a netlist of *look-up tables (LUTs)* [Cong and Ding, 1994] or LUT structures [Ray et al., 2012]. As with logic optimisation, the objective is to generate a circuit implementation with a minimal area or a minimal delay by considering the area and delay of the used technology blocks. Figure 1.3 shows the netlist of a 2-bit adder before and after logic optimisation, and after technology mapping. But, this small circuit is an easy target for logic synthesis, and it can even be easily processed by hand. Modern logic synthesis methods work with large circuits whose size varies from a few thousand to several million gates.

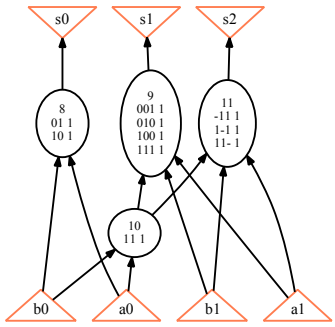
The evolution of logic synthesis is triggered mainly by the growth in the design size. The main role in this evolution belongs to the data structures for representing digital circuits along with the tools and algorithms



(a) Input gate-level netlist.



(b) The netlist after logic optimisation.



(c) The netlist after technology mapping.

Figure 1.3 – The logic synthesis process for a 2-bit adder. The initial gate-level netlist, which is comprised of 18 AND gates and has 5 levels, is shown on Figure 1.3a. Each internal (oval) node represents an AND gate and gives its ID number. Figure 1.3b shows the netlist after an algorithm for logic optimisation is run. Both area and delay are optimised, so the new netlist has only 11 AND gates and 4 levels. Finally, Figure 1.3c presents the network after technology mapping into 3-input LUTs. Each internal node represents a LUT and gives its ID number and SOP representation.

for their processing. The data structures that are used in this thesis are described in more detail in Section 2.2.

Once upon a time, minterm-based representations and algorithms were introduced

In order to produce an optimal two-level circuit implementation, logic synthesis started by proposing optimisation algorithms that use representations such as *truth tables* [Karnaugh, 1953; McCluskey, 1956; Quine, 1952; Veitch, 1952] and *sums of products (SOPs)* [Brayton et al., 1984]. Some algorithms operating with these representations are still used as building blocks of more complicated logic synthesis applications. For example, in technology mapping, the state-of-the-art

algorithms for *negation-permutation-negation (NPN)* classification use truth tables for classifying and matching functions [Huang et al., 2013; Petkovska et al., 2016a]. But in general, truth tables are impractical for functions with more than 16 inputs, whereas SOPs are inefficient for some practical circuits that are rich with XOR logic and for which the size of the minimal two-level representation is exponential in the number of primary inputs.

Transition to the more scalable DAG-based representations and algorithms

Due to these limitations, the attention was redirected to algorithms for multi-level logic optimisation [Brayton et al., 1987; Gregory et al., 1986] in which the circuits are represented as a *directed acyclic graph (DAG)* of logic gates. For example, ABC [ABC], which is a widely-used academic open-source software system for logic synthesis and formal verification, heavily relies on *and-inverter graphs (AIGs)* [Hellerman, 1963; Kuehlmann et al., 2002], but a rising interest exists also for the more recent *majority-inverter graphs (MIGs)* [Amarù et al., 2014a]. At the same time when the first multi-level logic optimisation algorithms appeared, algorithms that use *binary decision diagrams (BDDs)* started to emerge [Bryant, 1986; Coudert et al., 1993a]. The use of BDDs spread fast because their canonicity enables an efficient processing of two Boolean expressions once the BDDs are constructed. However, for some practical functions, such as multipliers, the BDD construction suffers from the *BDD memory explosion problem*—the BDD size has an exponential lower bound in the number of input variables—hence, using BDDs is often impractical [Bryant, 1991].

SAT solvers as a promising way to overcome the current limits of logic synthesis

In contrast, SAT solvers can be initialised in linear time to the circuit size and can immediately start solving the given problem. By not using a canonical representation, algorithms using SAT can avoid the exponential space blow-up of BDDs. Hence, the SAT-based algorithms are more scalable and offer better runtime and memory consumption in many cases. In the following section, we discuss the introduction and role of SAT solvers in logic synthesis in more detail. We also demonstrate that SAT solving has already been shown to be (1) a good option for implementing traditional logic synthesis algorithms, and (2) appropriate as an alternative or a replacement for BDDs. Moreover, by using the existing SAT-based algorithms, new applications can be easily built.

Example 1.1.1. Soeken et al. [2016] proposed a SAT-based algorithm for computing a dependency matrix that combines SAT-based algorithms from combinational equivalence checking [Mishchenko et al.,

2006b] and automatic test pattern generation [Larrabee, 1990], with some SAT-solver features such as incremental solving. Their algorithm also outperformed the BDD-based algorithm from ABC [ABC] for 13 out of 14 benchmarks—the BDD-based implementation did not solve 4 benchmarks due to a time limit and 5 benchmarks due to a memory limit, for 4 it has a higher runtime, and it is faster for only one benchmark that has a simple structure.

Exploring and analysing further which algorithms and features of SAT solvers are suitable specifically for logic synthesis and other EDA stages, as well as proposing new ways to implement and use SAT solvers, would ease the development of SAT-based logic synthesis applications and would enable problems, which were intractable before, to be solved.

Logic synthesis for combinational and sequential circuits

The algorithms for logic synthesis also differ depending on whether they optimise the combinational logic of the circuit or the sequential one. In this thesis, the proposed algorithms work only with combinational logic, whereas sequential circuits are handled by assuming all inputs and outputs of the sequential logic as primary outputs and inputs, respectively.

1.2 SAT Solving in EDA Applications

EDA algorithms based on SAT solvers are proposed for every part of the tool flow

SAT solvers have been deployed in almost all stages of EDA flows. Shifting towards SAT solving is encouraging because the improvement of the performance of state-of-the-art SAT solvers brings immediate benefit to the applications that use them. Furthermore, beside the regular improvement in SAT solvers, SAT algorithms have been proposed specifically for solving EDA problems and have been tuned to target distinct properties and structures of the digital circuits [Aloul et al., 2006; Goldberg and Novikov, 2002; Marques-Silva and Sakallah, 1999; Oh et al., 2004]. The interest in applying SAT to EDA began in the 1990s when SAT solvers were introduced in algorithms for routing in FPGAs [Nam et al., 1999; Wood and Rutenbar, 1998], placement [Devadas, 1989], timing analysis [Silva et al., 1998], fault diagnosis and logic debugging [Chen and Gupta, 1996; Smith et al., 2005], but also for logic synthesis and verification, as described next. The success of these initial algorithms showed the potential of SAT solvers in these stages of EDA, and the

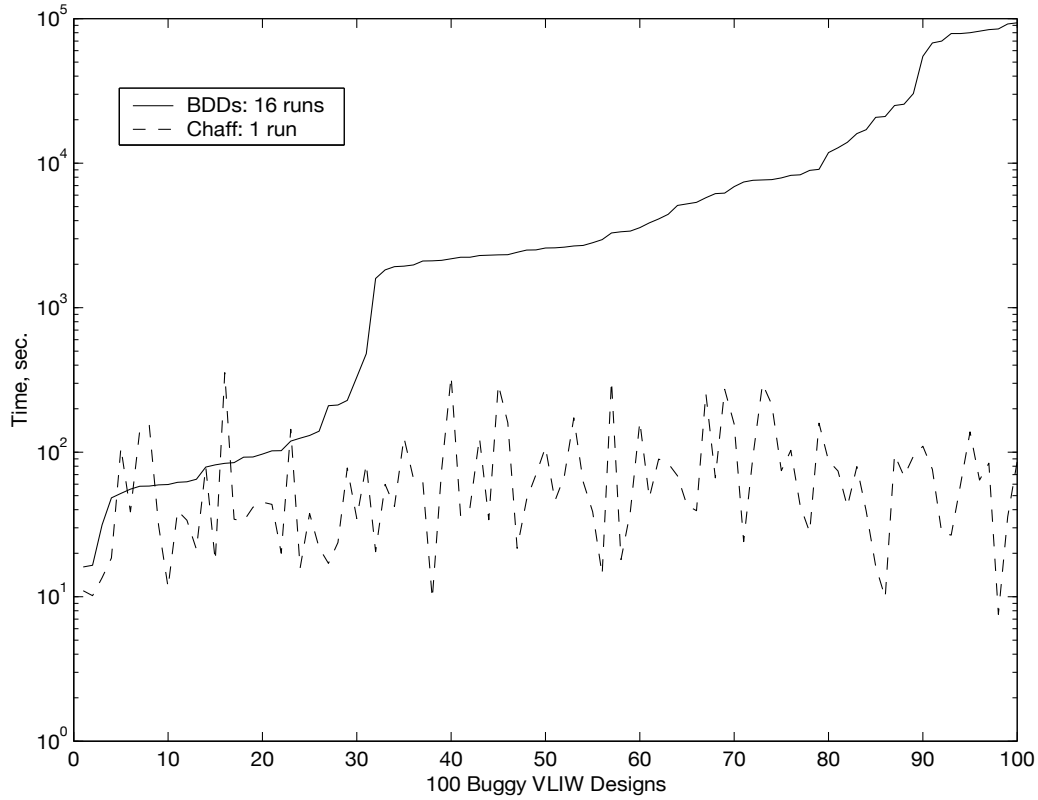


Figure 1.4 – Comparison of SAT and BDDs for functional verification. The benchmarks are Boolean formulas for verifying 100 buggy versions of a VLIW microprocessor. The used SAT solver Chaff [Moskewicz et al., 2001] evaluates only one correctness criterion. BDDs evaluate 16 easier criteria in parallel, and as soon as one of them finishes, the rest are terminated and the verification time of the one that finished is reported. The benchmarks are sorted in ascending order of their times for the BDD-based experiment [Velev and Bryant, 2003].

research in these areas is still active [Fraisse et al., 2016; Fujita and Mishchenko, 2014; Nadel and Ryvchin, 2016].

SAT as an alternative to BDDs: the success story from verification

The biggest motivation for developing SAT-based logic synthesis algorithms came from their success in formal verification [Velev and Bryant, 2003], especially in automatic test pattern generation [Fujita et al., 2015; Larrabee, 1990; Stephan et al., 1996], equivalence checking [Goldberg et al., 2001; Mishchenko et al., 2006b], and model checking [Biere et al., 1999; McMillan, 2003]. All these algorithms work also with Boolean logic, and are thus similar to logic synthesis. These applications proved that SAT-based methods can easily outperform existing BDD-based

methods, hence SAT solvers and algorithms using them became crucial components of most verification tools [Claessen et al., 2009].

Example 1.2.1. Velev et al. [2003] compared the runtime of procedures based on SAT solvers and BDDs by evaluating benchmarks that represent Boolean formulas for formal verification of correct and buggy versions of a *very long instruction word (VLIW)* microprocessor. In this case, first, they proved that the SAT solver Chaff [Moskewicz et al., 2001] has the best performance among other SAT solvers at that time. Next, they compared Chaff to a fast BDD-based method [Velev, 2000]. As Figure 1.4 shows, the difference between BDDs and Chaff is up to four orders of magnitude when verifying the buggy designs. When verifying correct designs, Chaff required 1.6 seconds, whereas the BDD-based method required 31.5 hours (about five orders of magnitude).

The wide and active use of SAT solvers in logic synthesis

This success in formal verification triggered a similar trend in logic synthesis. For example, more scalable SAT-based versions were proposed for the BDD-based algorithms for functional dependency [Jiang et al., 2010; Lee et al., 2007], functional decomposition [Lee et al., 2008; Lin et al., 2008], logic don't-care-based optimisation [Mishchenko and Brayton, 2005], and dependency matrix computation [Soeken et al., 2016c]. The algorithms based on SAT solvers are more scalable because they can process problems for which the size of BDDs is exponential, and they are faster when solving large and hard problems. However, the integration of SAT is not limited to replacing BDDs. SAT is shown efficient also for many other logic synthesis applications, including Boolean matching [Safarpour et al., 2006; Soeken et al., 2016a], technology mapping [Ling et al., 2005], and combinational delay optimisation [Soeken et al., 2017]. Due to the wide use of SAT in logic synthesis, but also in other stages of the EDA flow, SAT solvers became a standard part of the EDA tools, similarly to how BDD packages were integrated in the past. This facilitates the development and inclusion of new SAT-based algorithms in the flow.

1.3 Challenges of Exploiting SAT Solvers

Despite the wide use of SAT solvers in EDA and other applications, there are still many challenges that have to be addressed in order to facilitate and enable efficient implementation of some specific applications. In

this chapter, we present challenges that are among the most important for logic synthesis, and we therefore tackle most of them in this thesis.

Excessive runtime for hard problems, and unpredictable runtime in general

Although modern SAT solvers can often solve hard structured problems with over a million variables and several million constraints [Gomes et al., 2008], there are still some hard problems that cannot be solved within several hours of computing time. The required time for SAT solving depends mostly on the size and complexity of the problem, but other factors can also affect the ability to find a solution for a given problem. First, there are different *encoding* techniques that determine the translation into a propositional representation given as input to the SAT solver. Choosing an appropriate encoding is important as the quality of encoding often determines whether the problem is solvable or not [Björk, 2009]. Second, the SAT competitions have different competition tracks in order to accommodate different types of SAT solvers [Balyo et al., 2017]. Thus, although some SAT solvers might be unsuitable for solving some particular problems, others can demonstrate an exceptional runtime. An additional challenge is that it is impossible to predict the runtime and final result of a single call of the SAT-solving procedure. In this aspect SAT solvers are similar to BDDs, for which the termination time and the quality of results are unknown until the complete BDD is build. However, in this thesis, we show that in some cases SAT-based algorithms can be more scalable than their BDD-based versions. This scalability is achieved by structuring the algorithms to solve the problem gradually by issuing multiple easier calls of the SAT-solving procedure. The gradual problem solving also enables estimating the progress of the solving and the total required runtime.

Non-existent control over the returned satisfying assignment and proofs of unsatisfiability

For a given problem, SAT solvers return a canonical answer in terms of whether the problem is *satisfiable* or *unsatisfiable (UNSAT)*. However, the previously mentioned factors also affect the choice and quality of the received proof for the final evaluation, which is either a *satisfying assignment* or a *proof of unsatisfiability*, respectively. For example, when the problem is satisfiable, the choice of the satisfying assignment returned by the SAT solver depends on the encoding of the problem, the implementation of the SAT solver, and the underlying operating system. Moreover, in logic synthesis, the runtime and results are affected by the initial implementation of the circuit from which the SAT problem is derived. Thus, SAT solvers are often considered as non-canonical. This makes them unsuitable for applications requiring canonicity; hence,

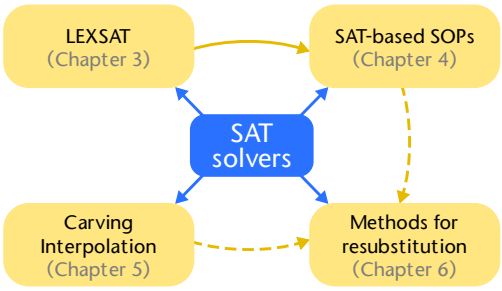


Figure 1.5 – Connection of the main contributions of the thesis. All algorithms are based on SAT solving. The LEXSAT algorithm is used for the SAT-based SOP generation to generate canonical SOPs. The methods used for resubstitution include, as first option, a method that is based on cube enumeration, similar to the one for SAT-based SOPs generation, and as second option, methods that are used as part of the carving interpolation algorithm. The last line of each box gives the chapter number that covers the corresponding topic.

such applications usually rely on canonical BDDs that also face scalability issues for some particular problems despite being the most scalable canonical representation. Similarly, even when canonicity is not required, different satisfying assignments and proofs of unsatisfiability can lead to results with different qualities in the applications that use them. In this thesis, we demonstrate that it is possible to achieve canonicity in SAT-based applications that utilise the returned satisfying assignments. We also propose a fast algorithm for this purpose. Moreover, despite having no control over the returned proofs of unsatisfiability, we obtain results with the wanted properties by calling the SAT-solving procedure multiple times with specific values for some inputs.

1.4 Thesis Contribution and Organisation

Facilitate the development of SAT-based logic synthesis applications

With this thesis, we facilitate the development of logic synthesis applications that use SAT solvers as a main engine of computation. We extend the set of available SAT-based methods with three novel approaches. Each *method* is an algorithm that functions as a key building block in different logic synthesis applications. Some proposed methods are not even restricted to logic synthesis and can be used as building blocks for other applications. Finally, we compare two SAT-based methodologies that are used for the resubstitution of a given function. Figure 1.5 illustrates the connection between the proposed methods and the compari-

son. Next, we give the outline of the thesis, as well as a short description of the chapters containing the main contributions.

First, **Chapter 2** comprises the background information to support the following chapters.

Fast generation of satisfying assignments in lexicographic order

Chapter 3 introduces a new fast algorithm [Petkovska et al., 2016b] for the lexicographic Boolean satisfiability problem, called LEXSAT, when the satisfying assignments are generated in a lexicographic order. Given a variable order, LEXSAT finds a satisfying assignment whose integer value under the given variable order is minimum (or maximum) among all satisfying assignments. If the formula has no satisfying assignments, LEXSAT proves it unsatisfiable, as does traditional SAT. We also propose methods that use the lexicographic properties of the assignments to further improve the runtime when generating consecutive satisfying assignments in lexicographic order. Finally, we promote the use of LEXSAT in a wide range of EDA applications, especially because it enables canonicity in SAT-based algorithms.

Progressive generation of canonical irredundant SOPs

Next, **Chapter 4** proposes an algorithm [Petkovska et al., 2017] that progressively generates canonical irredundant SOPs for completely and incompletely specified Boolean functions by using a SAT solver. For the first time, we can generate canonical SOPs using a SAT solver, which is enabled by the LEXSAT algorithm described in Chapter 3. Canonicity is a key component in applications, such as constraint solving and random assignment generation, which traditionally rely on methods based on BDDs. However, in contrast with BDDs, our algorithm can relax canonicity when it is not required in order to improve speed and scalability. Moreover, unlike the BDD-based methods for SOP generation, our progressive generation enables real time monitoring and early termination, as well as generation of partial SOPs for incremental applications.

Forcing the use of base functions during interpolation

Chapter 5 provides a new method [Petkovska et al., 2014] that can be particularly useful when rewriting circuits in some synthesis-based algorithms for which a dependency function h should be generated. The function h reimplements a given target function f as $f = h(G)$, where G represents a given set of base functions. The problem with the existing Craig interpolation, which can also provide the dependency function, is that it selects random base functions and, in particularly, omits some base functions potentially required for an optimal implementation of

the target function. Therefore, we propose a method, called *carving interpolation*, that forces a specific base function as a primary input of the dependency function by building the dependency function as a Shannon expansion of two constrained Craig interpolants. We also introduce an efficient method that iteratively imposes a predefined set of base functions.

Comparison of methods for resubstitution

Chapter 6 compares side-by-side two SAT-based methodologies for *resubstitution*, which replaces a function of a given *target node* using a *resubstitution function* that has other nodes as inputs, called *divisors*. Both methodologies were proposed as a substantial part of different algorithms for post-mapping optimisation and resynthesis, but they have never been compared to each other. Each methodology includes algorithms that can minimise the set of divisors while ensuring that the target node can be reimplemented with the minimised set, and can generate the resubstitution function. The first methodology combines methods that are also part of the carving interpolation presented in Chapter 5. First, it creates a minimal set of divisors by using a circuit structure similar to the one for obtaining an interpolant; it then generates a resubstitution function by using Craig interpolation. The second methodology is based on cube enumeration and is similar to the SAT-based SOP generation from Chapter 4. Its strength is that it directly provides the resubstitution function at the end of the minimisation. Lastly, we conclude that a hybrid approach that combines algorithms from the two methodologies represents an optimal solution for some applications.

Finally, **Chapter 7** concludes the thesis and provides ideas for future research that can build upon and extend the presented work.

2 Background Information

In this chapter, we provide background information on the basic terms and concepts that are used in the thesis. First, we introduce Boolean functions and their most commonly used representations. Then, we define satisfiability solving and describe the features available in modern SAT solvers that play a crucial role in the algorithms proposed in this thesis. Finally, we define the terminology associated with functional dependency, Craig interpolation and Shannon expansion, which are required for Chapter 5 and Chapter 6.

2.1 Boolean Function

Boolean function

For a variable v, a *positive literal* represents the variable v, whereas the *negative literal* represents its negation $\bar{v}$. A *cube*, or a product, c, is a Boolean product (AND, $\cdot$) of literals, $c = l_1 \cdot \cdots \cdot l_k$. If a variable is not represented by a negative or a positive literal in a cube, then it is represented by a *don't-care (−)*, meaning that it can take both values 0 and 1. A cube with i don't-cares, covers 2^i minterms. A *minterm* is the smallest cube in which every variable is represented by either a negative or a positive literal.

Example 2.1.1. Consider the set of variables $X = (x_1, x_2, x_3)$. A cube $-x_2\bar{x}_3$, which can be simply written as $x_2\bar{x}_3$, covers the two minterms $\bar{x}_1 x_2 \bar{x}_3$ and $x_1 x_2 \bar{x}_3$ because the variable x_1 is represented by a don't-care.

Assignment

Each minterm is associated with a given input assignment. An *assignment* to a finite set of Boolean variables, $X = (x_1, \ldots, x_k)$, is the mapping

$X \rightarrow \{0,1\}^k$. In the minterm, a variable is represented with the positive or negative literal if it assumes a value of 1 or 0, respectively.

Example 2.1.2. Consider the set of variables $X = (x_1, x_2, x_3)$. An assignment 010 denotes that $x_1 = 0$, $x_2 = 1$, and $x_3 = 0$. The minterm associated with this assignment is $\bar{x}_1 x_2 \bar{x}_3$.

Incompletely specified Boolean function and support sets

Let $f(X) : B^n \rightarrow \{0,1,-\}$, $B \in \{0,1\}$, be an *incompletely specified Boolean function* of n variables $X = \{x_1, \ldots, x_n\}$. The *support set* of f is the subset of variables that determine the output value of the function f. The *on-set* of f is defined by the set of minterms for which f evaluates to 1. Similarly, the *off-set* and the *don't-care-set* of f are defined by the minterms for which f evaluates to 0 and don't-care, respectively. In a multi-output function $F = \{f_1, \ldots f_m\}$, each output f_i, $1 \leq i \leq m$, has its own support set, on-set, off-set and don't-care-set associated with it.

Completely specified Boolean function

Similarly, $f(X) : B^n \rightarrow \{0,1\}$, $B \in \{0,1\}$ is a *completely specified Boolean function* of n variables $X = \{x_1, \ldots, x_n\}$. For completely specified functions, the minterms belong either to the *on-set* or to the *off-set* of f.

Example 2.1.3. Consider the function $f(x_1, x_2, x_3) = (x_1 + x_2)\bar{x}_3$. For the assignment 010, the function $f(x_1, x_2, x_3)$ evaluates to 1. Thus, its corresponding minterm $\bar{x}_1 x_2 \bar{x}_3$ belongs to the on-set of f. On the contrary, the minterm $\bar{x}_1 \bar{x}_2 x_3$ belongs to the off-set of f, because the function evaluates to 0 for the corresponding assignment 001.

Primary inputs and outputs

In a multi-output Boolean function $F = \{f_1, \ldots f_m\}$ defined over a set of input variables $X = \{x_1, \ldots, x_n\}$, the input variables x_i, $1 \leq i \leq n$, are called *primary inputs (PIs)*, and the variables of the outputs f_j, $1 \leq j \leq m$, represent the *primary outputs (POs)*. The PIs and POs represent the external connections of the network.

2.2 Representation of Boolean Functions

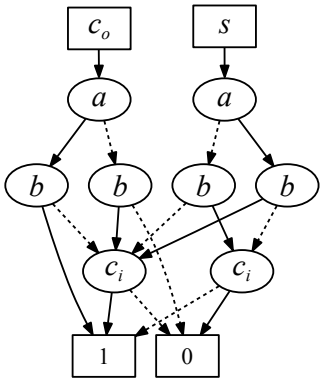
In this section, we define four possibilities for representing Boolean functions that are used in the following chapters of this thesis. First, we introduce the minterm-based representations truth tables and sums of products. Then, we introduce two representations based on *directed acyclic graphs (DAGs)*: binary decision diagrams and and-inverter graphs. Figure 2.1 illustrates the different representations through the sum and carry function of a full adder.

c_i	a	b	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

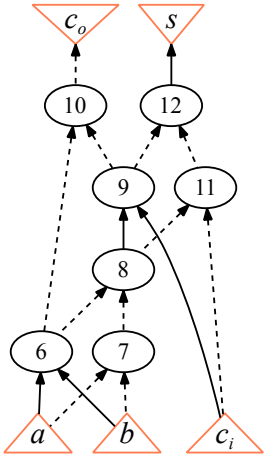
(a) A truth table.

c_o	s
-11	001
1-1	010
11-	100
	111

(b) An SOP.



(c) A BDD.



(d) An AIG.

Figure 2.1 – Representation of the Boolean functions of a full adder. The carry output is represented with the function $c_o = ab + c_i(a \oplus b)$, and the sum output is represented with the function $s = a \oplus b \oplus c_i$.

Canonicity

Some representations of Boolean functions are canonical. A *canonical representation* is a unique representation for a function under certain conditions. For example, the BDD structure of a Boolean function depends only on the input variable order and is independent of the original representation of the function. Canonicity is an important feature for some applications, such as functional equivalence checking.

2.2.1 Truth Tables

Truth tables

A single-output Boolean function f with n inputs can be represented with its *truth table*, which is a string composed of 2^n bits. The i-th bit of the truth table gives the output value of the function f when its inputs

are assigned to the minterm defined with the binary digits of the n-bit binary number i. For example, the bit in position 0 gives the value of the function when all PIs are assigned to 0. The bitstring representing the truth table can be observed as the binary expansion of a non-negative number $t_f \in [0, 2^n)$. For example, the truth table of any 5-input function can be encoded with a 32-bit integer. As shown on Figure 2.1a, 8 bits are required to represent the truth table of each full-adder output that has 3 inputs.

Example 2.2.1. The truth table of the two-input function $f(x_1, x_2) = x_1 \cdot x_2$ is 0001 and $t_f = 7$. The three-input function $g(x_1, x_2, x_3) = x_1 \cdot x_2 + \bar{x}_3$ has the truth table 10101011 and $t_g = 171$.

Pros and cons of truth tables

The truth table of a given function is always canonical. They provide a simple way to represent and manipulate Boolean functions. However, they are not practical for functions that have more than 16 inputs, due to their exponential growth in size in terms of the number of inputs.

2.2.2 Sums of Products

Sums of products

Another more compact minterm-based representation is the two-level *sum of products (SOP)*, which is a Boolean sum (OR, +) of cubes, $S = c_1 + \cdots + c_k$. Assume that a Boolean function f is represented as an SOP S_f. The SOP can be also represented as the set of cubes that comprise the sum of cubes by representing the positive, negative and don't-care variables of the cubes with 1, 0 and −, respectively.

Example 2.2.2. Consider the functions f and g from Example 2.2.1. The on-set SOP of the two-input function f, $S_f = x_1 x_2$, can be also written as $S_f = \{11\}$. The on-set SOP $S_g = x_1 x_2 + x_1 \bar{x}_3 + \bar{x}_1 \bar{x}_3$ of the three-input function g the can be also written as $S_g = \{11-, 1-0, 0-0\}$.

Prime cubes and irredundant SOPs

A cube is *prime*, if no literal can be removed from the cube without changing the value that the cube implies for f. A cube that is not prime, can be *expanded* by substituting at least one literal with a don't-care. The SOP is *irredundant* if each cube is prime and no cube can be deleted without changing the function.

Pros and cons of SOPs

Typically, a canonical SOP refers to the canonical disjunctive normal form that represents a sum of minterms and is canonical for a given

function, but these SOPs are redundant. However, for a given function, some algorithms [Minato, 1992] can generate canonical irredundant SOPs that depend only on the input variable order. The disadvantage of SOPs is that their two-level representation leads to an ineffective manipulation of large logic nodes. Moreover, they are inefficient for some arithmetic, error correcting, and telecommunication circuits that are rich with XOR logic. For example, Figure 2.1b shows that the SOP of the sum output, which implements a 3-input XOR gate, is composed of four complete cubes (i.e., minterms), and the carry output is represented with 3 cubes, each composed of only 2 literals.

2.2.3 Binary Decision Diagrams

Binary decision diagrams

One of the fundamental representations of Boolean functions are the BDDs that were first introduced by Lee et al. [1959] and Akers [1978]. A *binary decision diagram (BDD)* is a DAG with decision nodes representing the input variables of the Boolean function and terminal nodes representing either a constant 1 or 0. One of the decision nodes is the root node of the DAG. Each decision node has two output edges labelled with 0 and 1, and it actually represents a 2-to-1 multiplexer. Any path from the root node to any terminal 1 or 0 node defines a cube for which the represented Boolean function evaluates to 1 or 0, respectively. For example, Figure 2.1c shows the BDD of the full adder in which some decision nodes are shared among the two outputs.

Refinements of BDDs

Several refinements of the BDDs have been proposed in the literature [Wegener, 2000], such as transformation binary decision diagrams (TBDDs) [Goldberg et al., 1997], biconditional binary decision diagrams (BBDDs) [Amarù et al., 2013], and zero-suppressed decision diagrams (ZDDs) [Mishchenko, 2014]. In general, the term BDD refers to the most commonly used form, called *reduced ordered binary decision diagram (ROBDD)* [Bryant, 1986], that is canonical for a particular function and variable order.

Variable order in BDDs

A BDD is ordered [Bryant, 1986] if the input variables appear in the same order on all paths from the root to the terminal nodes. The size of the BDD for a given function highly depends on the chosen variable order. As finding the best variable order is NP-hard problem [Bollig and Wegener, 1996], many efficient heuristics were developed [Rice and Kulhari, 2008].

Pros and cons of BDDs

BDDs are typically canonical for a given variable order. However, due to their canonicity, the size of the ordered BDDs is exponential, regardless of the variable order for some practical functions where the same inputs serve as both control and data [Bryant, 1991], which makes BDDs often impractical for use. An example for such functions are multipliers, floating-point alignment units, and the hidden weighted bit function [Knuth, 2009]. There are some refinements that can relax canonicity, but they are usually harder to manipulate.

2.2.4 And-Inverter Graphs

Boolean networks

Another graph-based option for the representation of a Boolean function is a multi-level Boolean network. The *Boolean network* is a DAG in which the internal nodes correspond to components implementing a given logic function, such as gates or LUTs, and the directed edges correspond to wires connecting these components. Incoming edges of a node are called *fanins* and outgoing edges are called *fanouts*. The PIs of the network are represented as source nodes without fanins, and the POs are represented as sink nodes without fanouts. A *transitive fanin (TFI)* or *transitive fanout (TFO)* cone of a node is a subset of network nodes that are reachable through the fanin or fanout edges of the node, respectively. A *cut* of a node n, called the *root*, is a set of nodes, called *leaves*, such that each path from a PI to n passes through at least one leaf. A cut is *k-feasible* if the number of leaves is less or equal to k.

And-Inverter Graphs

One of the simplest and most commonly used Boolean networks is the *and-inverter graph (AIG)* that is composed of two-input AND gates and inverters [Brayton and Mishchenko, 2010; Hellerman, 1963; Kuehlmann et al., 2002]. All internal nodes of the DAG are two-input AND gates, whereas inverters are represented as attributes on the edges. The AIGs have a special constant node that has no inputs, either representing a Boolean 1 or 0. Any logic network can be converted to an AIG implementing the same Boolean function of the POs in terms of the PIs.

Structural hashing

One of the most important methods for AIGs is *structural hashing*. It ensures that no two AND nodes in the AIG have the same two fanins, including the inverters. Structural hashing is usually applied during the creation of the AIG. It reduces the number of nodes in it and increases the logic sharing in the network. For example, Figure 2.1d shows the structurally hashed AIG of the full adder.

AIGs enable short runtimes and high-quality results for synthesis, mapping and verification due to their simplicity and flexibility, hence it is a common data structure in many academic and industrial systems for logic synthesis and verification. Their main disadvantage is that they are not canonical.

Pros and cons of AIGs

2.3 Boolean Satisfiability

A disjunction (OR, +) of literals forms a *clause*, $t = l_1 + \cdots + l_k$. A *propositional formula* is a logic expression defined over variables that take values in the set {0, 1}. To solve a SAT problem, a propositional formula is converted into its *conjunctive normal form (CNF)* as a conjunction (AND, ·) of clauses, $F = t_1 \cdot \cdots \cdot t_k$. Algorithms such as the Tseitin transformation [Tseitin, 1983] convert a Boolean function into a set of CNF clauses.

Conjunctive normal form

Definition 1. A *Boolean satisfiability (SAT) problem* is a decision problem that takes a propositional formula in CNF form and returns that the formula is satisfiable if there is an assignment of the variables from the formula for which the CNF evaluates to 1. Otherwise, the propositional formula is *unsatisfiable (UNSAT)*.

Boolean satisfiability

A programme that solves SAT problems is called a *SAT solver*. SAT solvers provide a *satisfying assignment* when the problem is satisfiable. Otherwise, modern SAT solvers, such as MiniSat [Eén and Sörensson, 2003], can provide a *proof of unsatisfiability*, also called a *refutation proof*. To define refutation proof, we need to first define the resolution principle.

SAT solvers

Definition 2. Let $c_1 = x + R_1$ and $c_2 = \bar{x} + R_2$ be any two clauses, such that if there is a literal x in c_1, then its complement, $\bar{x}$, is a literal in c_2. The *resolution principle* says that the *resolvent* of the clauses c_1 and c_2 is the disjunction $R_1 + R_2$, given that $R_1 + R_2$ is non-tautological. The literal x is called a *pivot variable*.

Resolution principle

Definition 3. A *refutation proof* Π of a set of clauses C is a directed acyclic graph (V_Π, E_Π), where E_Π is set of edges connecting the vertices with their predecessor vertices, and V_Π, the set of vertices, presents a set of clauses such that

Refutation proof

- for every vertex $c \in V_\Pi$, c is either a *root clause*, such that $c \in C$, or c is an *intermediate clause* and represents the resolvent of its two predecessors c_1 and c_2, and
- the unique *leaf vertex* is an empty clause.

Essentially, the resolvent is a necessary clause for the conjunction of the original clauses to be satisfiable, and the refutation proof shows how to pair clauses in order to derive necessary conditions for satisfiability until the result is a contradiction. In Section 2.5, we show how the refutation proof is used for building an interpolant.

Example 2.3.1. Consider the UNSAT set of clauses of Figure 2.4, which is further discussed in Section 2.5: the refutation proof asserts that the last two clauses can only be satisfied if $b = 0$ and, therefore, can be replaced with their resolvent $(a + c)$. Next, the clauses $(\bar{a} + d)$ and $(a + c)$ can be replaced by the resolvent $(c + d)$, and so on, until we obtain the leaf clause 0, which proves that the original set of clauses is UNSAT.

Incremental SAT solving with assumptions

Modern SAT solvers can also determine the satisfiability of a problem under given assumptions. *Assumptions* are propositions that are given as input to the SAT solver for a specific single invocation of the SAT solver and have to be satisfied for the problem to be SAT. The process of determining the satisfiability of a problem under given assumptions is called *incremental SAT solving*. Some SAT solvers support an internal stack of assumptions, which enables adding and removing assumptions between consecutive SAT calls via a *push/pop mechanism*. With this, the state of the SAT solver is preserved between incremental runs, whereas incremental runs themselves enable reusing learned clauses from previous calls of the SAT-solving procedure. Thus, both incremental SAT solving with assumptions and incremental adding/removing of assumptions lead to flexibility and efficiency in SAT-based applications.

Example 2.3.2. For the function $f(x_1, x_2, x_3) = (x_1 + x_2)\bar{x}_3$, which is satisfiable for the following assignments of the inputs $\{010, 100, 110\}$, a SAT solver without assumptions can return any of the given assignments. But, if we give as input to the SAT solver the assumption $x_1 = 1$, then it returns either 100 or 110, because those two assignments satisfy the given assumption.

A set of assumptions for UNSAT

The subset of root clauses from the proof of unsatisfiability, which belong to the original SAT problem, are called *UNSAT core*. Most appli-

cations do not require the full proof of unsatisfiability nor the complete UNSAT core. Thus, when the problem is UNSAT under a given set of assumptions, some modern SAT solvers can return an abstraction of the core as a subset of assumptions used in the proof of unsatisfiability, which we call *a set of assumptions for UNSAT*. For example, the SAT solver MiniSAT [Eén and Sörensson, 2003] provides a dedicated procedure, called "analyze_final", that returns a set of assumptions for UNSAT in the form of a final conflict clause [Eén et al., 2010]. This feature is very practical because it enables calls that evaluate to UNSAT to be as useful as the ones that evaluate to satisfiable, without logging and traversing the complete proof of unsatisfiability.

Example 2.3.3. Assume that a SAT solver was initialised with the function f from Example 2.3.2. If we call the SAT-solving procedure with the assumptions $x_1 = 1$ and $x_3 = 0$, then the problem evaluates to UNSAT as there is no satisfying assignment that satisfies the given assumptions. If we ask for the set of assumptions for UNSAT, the SAT solver will return either the complete set $\{x_1, \bar{x}_3\}$ or the subset $\{\bar{x}_3\}$ depending on the SAT-solving algorithm.

2.4 Functional Dependency

Functional dependency

The check for functional dependency ensures that a given function f can be expressed by a given set of base functions G. Additionally, the necessary and sufficient condition of functional dependency helps to define the construction of the circuits from which an interpolant is built.

Definition 4. A function $f(X)$, defined over the variable vector $X = (x_1, \ldots, x_m)$, *functionally depends* on a set of Boolean functions $G = \{g_1(X), \ldots, g_n(X)\}$ if there exists a Boolean function h such that $f(X) = h(g_1(X), \ldots, g_n(X))$ [Jiang et al., 2010].

The functions f, g_i, and h are called *target function*, *base functions*, and *dependency function*, respectively.

Example 2.4.1. Consider the carry function of a 2-bit adder c_1

$$c_1 = (a_1 \cdot b_1) + (a_0 \cdot b_0 \cdot (a_1 + b_1)),$$

which is defined over the variable vector $X = (a_0, b_0, a_1, b_1)$, as a target function. Given is a set of base functions $G = \{g_1, g_2, g_3, g_4\}$, where

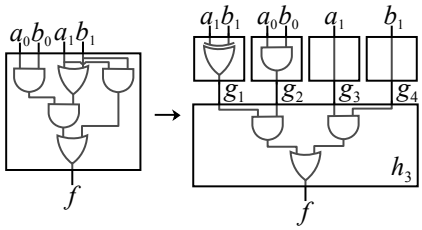


Figure 2.2 – Representation of a target function with a set of base functions. The target function c_1 from Example 2.4.1 is implemented using the set of base functions G and the dependency function h_3. A dependency function h_i exists if and only if the target function c_1 functionally depends on the set of base functions G.

$$g_1 = a_1 \oplus b_1,$$
$$g_2 = a_0 \cdot b_0,$$
$$g_3 = a_1, \text{ and}$$
$$g_4 = b_1.$$

As the target function c_1 can be rewritten as any of the dependency functions

$$h_1 = (g_3 \cdot g_4) + (g_2 \cdot (g_3 + g_4)),$$
$$h_2 = (g_3 \cdot g_4) + (g_2 \cdot (g_3 \oplus g_4)), \text{ or}$$
$$h_3 = (g_3 \cdot g_4) + (g_2 \cdot g_1),$$

it follows that f functionally depends on the set of base functions G. Figure 2.2 shows the target function c_1 implemented using the set G and its dependency function h_3.

Types of base functions

A base function $g_i \in G$ is an *essential* base function, if f functionally does not depend on G when g_i is removed from the set G. Otherwise, g_i is an *auxiliary* base function. In Example 2.4.1, given c_1 as target function and the set $G = \{g_1, \ldots, g_4\}$, the base functions g_2, g_3 and g_4 are essential, whereas g_1 is an auxiliary base function.

Necessary and sufficient condition for functional dependency

Next, we give the necessary and sufficient condition for functional dependency that is used to construct the circuit for checking if a function f functionally depends on a set of base functions G.

Theorem 1. Let $f(X)$ be a target function and let G be a set of Boolean functions $G = \{g_1(X), \ldots, g_n(X)\}$, all defined over the variable vector X.

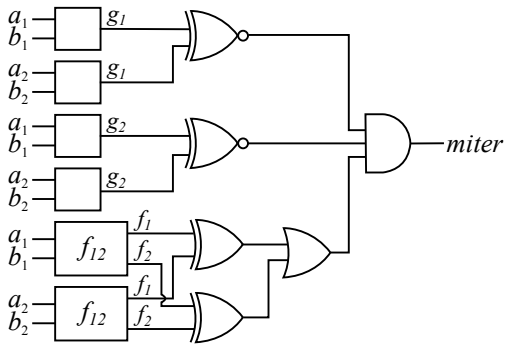


Figure 2.3 – A miter for checking if a target function depends on a set of base functions. The miter constructed for checking if the function f_{12}, with two outputs f_1 and f_2, and two inputs a and b, functionally depends on the set of base functions $G = \{g_1, g_2\}$. Functional dependency exists if and only if the miter evaluates to 0 for any two assignments of the primary inputs, that is, the miter is UNSAT.

For any two assignments P and Q of X, when $f(P) \neq f(Q)$, then the set G contains at least one base function $g_i(X)$, for $i = 1, \ldots, n$, such that $g_i(P) \neq g_i(Q)$, if and only if, the function f functionally depends on the set G [Jiang and Brayton, 2004].

Constructing a miter for functional dependency check

Following Theorem 1, to check if a function f functionally depends on a set of base functions G, we construct a *miter* circuit that we can transform to CNF clauses and give to a SAT solver. The miter evaluates to 1 if and only if two assignments for the primary inputs P and Q exist, for which each $g_i \in G$ evaluates to the same value and at least one output of the function f evaluates to a different value. The output that evaluates to a different value cannot be represented as a function of G, and there is no functional dependency. Otherwise, if the miter evaluates to 0 for all possible assignments of the primary inputs, then f functionally depends on G. As an example, Figure 2.3 shows a miter constructed for a multiple-output target circuit.

2.5 Craig Interpolation

Craig interpolation

In this section, we present the Craig interpolation theorem that was first proved by W. Craig [1957], and we show the method for constructing interpolants proposed by McMillan [2003].

Theorem 2. Let (A, B) be a pair of sets of clauses, such that $A \cdot B$ is unsatisfiable. Then, there exists a formula P such that

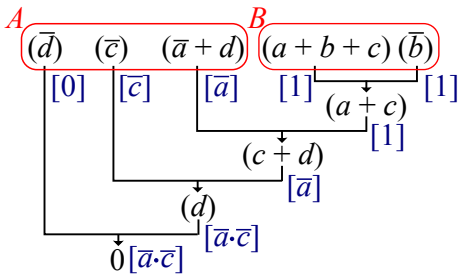


Figure 2.4 – Computing an interpolant from a refutation proof by using the McMillan's algorithm. The given refutation proof is for the pair of clauses $A = (\bar{d}) \cdot (\bar{c}) \cdot (\bar{a}+d)$ and $B = (a+b+c) \cdot (\bar{b})$. The intermediate clauses are derived using the resolution principle defined in Section 2.3. In square brackets, we give the Boolean formulas $p(c)$ assigned to each clause and the one assigned to the leaf clause, $p(0) = \bar{a} \cdot \bar{c}$, is the interpolant.

- A implies P,
- $P \cdot B$ is unsatisfiable, and
- P refers only to the common variables of A and B [Craig, 1957].

The formula P represents an *interpolant* of A and B. Given the pair (A, B) and their refutation proof, a procedure called *interpolation system* constructs an interpolant in linear time and space in the size of the proof [McMillan, 2003; Pudlák, 1997] .

McMillan's interpolation system

Next, we explain the construction of the interpolant through McMillan's system [McMillan, 2003].

Definition 5. Let (A, B) be a pair of clause sets and let Π be their refutation proof. To all clauses c from Π, we assign a Boolean formula $p(c)$, such that

- if c is a root clause, and
  - if $c \in A$, then $p(c)$ is the disjunction of c's global literals, whose variable appear in both A and B, or
  - if $c \notin A$, then $p(c) = 1$;
- and if c is an intermediate clause, then let c_1 and c_2 be the predecessor clauses of c, and let x be their pivot variable. Then,
  - if $x \in B$, then $p(c) = p(c_1) \cdot p(c_2)$, or
  - if $x \notin B$, then $p(c) = p(c_1) + p(c_2)$.

The Π-interpolant of (A, B) is the Boolean formula assigned to the leaf clause $p(0)$ [McMillan, 2003].

The Boolean circuit representing the interpolant is constructed by substituting the intermediate vertices and the leaf with gates corresponding to the executed operation between their predecessors. Figure 2.4 shows how the interpolant for $A = (\bar{d}) \cdot (\bar{c}) \cdot (\bar{a} + d)$ and $B = (a + b + c) \cdot (\bar{b})$ is constructed by following McMillan's algorithm.

2.6 Shannon Expansion

Shannon expansion

Shannon expansion [Shannon, 1949] is a fundamental theorem used for simplification and optimisation of logic circuits.

Theorem 3. Any Boolean function f defined over a variable vector $X = (x_1, \ldots, x_n)$, can be written in the form

$$f = \bar{x}_i \cdot f_{\bar{x}_i} + x_i \cdot f_{x_i},$$

where $f_{\bar{x}_i} = f(x_1, \ldots, 0, \ldots, x_n)$ and $f_{x_i} = f(x_1, \ldots, 1, \ldots, x_n)$.

Cofactors of a circuit

The expressions $f_{\bar{x}_i}$ and f_{x_i} represent, respectively, a *negative* and *positive cofactor* of f with respect to the *control variable* x_i.

Example 2.6.1. Consider the function $f(x_1, x_2, x_3) = (x_1 + x_2)\bar{x}_3$. The cofactors of f with respect to the variable x_3 are $f_{\bar{x}_3} = f(x_1, x_2, 0) = x_1 + x_2$ and $f_{x_3} = f(x_1, x_2, 1) = 0$. Thus, the function can be reconstructed as

$$f(x_1, x_2, x_3) = \bar{x}_3 \cdot f_{\bar{x}_3} + x_3 \cdot f_{x_3} = \bar{x}_3(x_1 + x_2) + x_3 \cdot 0 = (x_1 + x_2)\bar{x}_3.$$

3 Fast Generation of Lexicographic Satisfying Assignments

LEXSAT returns the smallest or greatest satisfying assignment

Lexicographic satisfiability (LEXSAT) is a decision problem similar to the SAT problem: for a given SAT formula it returns a satisfying assignment, if the problem is satisfiable, or otherwise it returns UNSAT. The only difference is that SAT can return *any* satisfying assignment, whereas LEXSAT returns deterministically the one whose integer value under a given variable order is the minimum (or maximum) among all satisfying assignments. The assignments with the minimum and maximum integer values are called the *lexicographically smallest* and *lexicographically greatest* assignment, respectively. For simplicity, we assume that LEXSAT always generates the lexicographically smallest assignment, but the same principles apply when generating the lexicographically greatest one.

Example 3.0.1. Assume a 4-input function $f(x_1, x_2, x_3, x_4)$ with the satisfying assignments for the inputs $\{0001, 0101, 1010, 1011, 1101\}$. SAT can return any of the given assignments, whereas LEXSAT always returns either the lexicographically smallest assignment 0001 or the lexicographically greatest assignment 1101, depending on the user preference.

Two LEXSAT algorithms by Knuth

Knuth [2015] mentions two implementations of an algorithm for generating satisfying assignments in a lexicographic order. The first one calls a SAT solver multiple times [Knuth, 2015, Ex. 7.2.2.2-109]: the first call generates a satisfying assignment that is iteratively minimised with the successive SAT calls. The second one implements the same concept by modifying the decision heuristic of the SAT solver in order to per-

This chapter is based on the work of a paper published at the 2016 International Conference on Computer Aided Design [Petkovska et al., 2016b].

form decisions on the input variables in a given order, whereas for the other variables decisions can be performed in any order [Knuth, 2015, Ex. 7.2.2.2-275]. However, Knuth does not evaluate the performance of these two algorithms.

The implementation that uses the SAT solver repeatedly is more robust than the one integrated in the SAT solver

Nadel and Ryvchin [2016] propose, independently, Knuth's LEXSAT algorithm, which they call `OBV-BS`, in the context of *satisfiability modulo theories (SMT)* solving. They also propose another algorithm integrated in a SAT solver. Their results are two-fold: First, the two proposed algorithms are faster than algorithms based on SMT solvers. Second, they show that the `OBV-BS` algorithm, which uses the SAT solver repeatedly, is slower than the one integrated in the SAT solver but it is more robust: it succeeds in finding solutions for difficult instances for which the integrated one exceeds the given time limit. A generalisation of Knuth's algorithm is also proposed by Marques-Silva et al. [2011].

A novel algorithm for fast generation of LEXSAT assignments

In this chapter, we propose our first SAT-based method—a scalable and fast LEXSAT algorithm that also repeatedly uses the SAT solver. But, instead of starting from a satisfying assignment that is iteratively minimised, we start from a potential assignment that is the lexicographically smallest assignment that might be satisfying. Then, for each variable, we iteratively either confirm that its assignment is identical to the one in the lexicographically smallest satisfying assignment, or we increase it, if possible. To achieve a good performance, we also propose a version of the algorithm that is based on the concept of binary search. Moreover, we propose methods that use the lexicographic properties of the assignments to further improve the runtime when consecutive satisfying assignments are generated in lexicographic order, which is required in applications such as the canonical SAT-based SOP generation presented in Chapter 4. For all algorithms, we propose to use incremental SAT solving to mimic the alternative implementation that modifies the SAT solver, which leads to a good performance while maintaining the SAT solver unmodified for general use. The experimental results show that our algorithm is faster than the first algorithm proposed by Knuth [2015], both for generation of non-consecutive and consecutive LEXSAT assignments.

LEXSAT can enable canonicity in EDA applications

We also identified that LEXSAT can be especially useful for a variety of EDA applications that require canonicity. One example for such an application is our generation of canonical SOP using SAT solvers, which

is presented in Chapter 4. In addition, Section 3.2 gives a summary of the applications that already use LEXSAT, as well as potential applications that can benefit from it in the future.

The widespread application of LEXSAT

LEXSAT can be applied in other domains for applications that require canonicity, the smallest or largest satisfying assignment, or consecutive satisfying assignments. As presented in Section 3.3, the variations of the SAT problem MAX-SAT, AllSAT, and #SAT can also be solved by a LEXSAT-based algorithm. These algorithms are used in a wide range of domains and expand the potential applicability of LEXSAT. Moreover, by using the concept of LEXSAT, we propose an algorithm LEX-UNSAT that can enable canonicity when the problem is UNSAT.

In the rest of the chapter, in Section 3.1, we give the formal definition of LEXSAT. In Section 3.2 and Section 3.3, we explain how we can use LEXSAT for EDA applications and for solving other SAT problems, respectively. In Section 3.4, we describe two versions of our algorithm and the methods for improving the runtime. We present our experimental setup and results in Section 3.5. In Section 3.6, we argue that our implementation with repetitive SAT calls is expected to be as efficient as an implementation that modifies the SAT solver. We conclude this chapter and present ideas for future work in Section 3.7. The required background information is provided in Section 2.1 and Section 2.3.

3.1 Lexicographic Boolean Satisfiability

Definition of LEXSAT

The *lexicographic satisfiability (LEXSAT) problem* is a variation of the SAT problem that takes a propositional formula in CNF form and a given variable order, and returns a satisfying variable assignment whose integer value under the given variable order is minimum (maximum) among all satisfying assignments. If the formula has no satisfying assignments, LEXSAT proves it unsatisfiable.

State-of-the-art LEXSAT algorithm

Recall that Knuth [2015] proposes two solutions for generating a LEXSAT assignment. In this chapter, we compare our algorithms to the first solution that calls the SAT solver multiple times. Assuming a function $f(x_1,\ldots,x_n)$, with the first call, the algorithm generates an *initial satisfying assignment* $a_1 \ldots a_n$, or terminates if the problem is UNSAT. Then, if the problem is SAT, it minimises the assignment iteratively. For this, a pointer d is set to 0 before the first iteration, and later points to the

next variable that is assigned to 1 and can be flipped to 0 in order to decrease the assignment. Assignments for the variables x_i for $1 \leq i < d$ are considered to be fixed. Thus, to minimise the assignment, first, d is set to the index of the next variable that is assigned to 1. If $d > n$, then no variable in the assignment can be flipped, and the algorithm returns $a_1 \dots a_n$. Otherwise, using the assumption mechanism, the SAT solver is called again with the assumptions $x_i = a_i$, for $1 \leq i < d$, and $x_d = 0$. If the problem is SAT, the assignment $a_1 \dots a_n$ is updated with the newly received assignment; otherwise, the old assignment is kept. Finally, it performs another iteration for minimisation to find the next non-fixed 1 to be flipped.

Example 3.1.1. For a function $f(x_1, x_2, x_3, x_4, x_5)$, assume that the assignment 00101 is received with the first SAT call. Then, in the first iteration for minimisation, the pointer d is set to 3, because x_3 is the first variable that can be flipped from 1 to 0. Next, the SAT solver is called with the assumptions $x_1 = 0$, $x_2 = 0$, $x_3 = 0$. If the problem is UNSAT, the value of x_3 remains 1, because there is no satisfying assignment that satisfies the given assumptions (i.e., that starts with 000); thus, the old assignment is kept and in the second iteration for minimisation d is set to 5. Otherwise, assuming that the SAT solver returns the assignment 00010, it is considered as a potential assignment in the second iteration, so $d = 4$.

3.2 LEXSAT for EDA Applications

NPN classification for large functions

Although LEXSAT has emerged only recently, it has already been shown useful for several EDA applications. For example, Soeken et al. [2016] show that LEXSAT enables heuristic NPN classification of large functions with up to 194 variables. The same heuristic algorithm was previously limited to functions with up to 16 variables, for which truth tables could be computed [Huang et al., 2013]. However, LEXSAT enables comparing two functions $f(X)$ and $g(X)$, where X is a set of variables $X = \{x_1, \dots, x_n\}$, by solving the SAT problem defined with the formula $f(X) \oplus g(X)$, which is commonly used for equivalence checking of two circuits [Brand, 1993]. Thus, if the problem is UNSAT, then the functions are identical, i.e., $f = g$. Otherwise, if the problem is satisfiable, then the smallest LEXSAT assignment defines the most significant bit for which the truth tables of f and g differ. By simulating this assignment for one

of the functions, it can be determined which one has a truth table with a larger value. For example, if LEXSAT returns an assignment A and if $f(A) = 0$, then $f < g$.

Correcting a cell placement during physical design

LEXSAT is also used for fixing a cell placement during the physical design stage of an industrial EDA flow [Nadel and Ryvchin, 2016]. By finding the maximal value of a bit-vector, which encodes that a potential violation is solved, a fixer tool generates a placement that has as few violations as possible while giving preference to fixing high-priority violations that are encoded with the most significant bits of the bit-vector.

SAT-based generation of canonical SOPs

Another example is the work presented in Chapter 4, where LEXSAT enables generation of canonical SOPs by using a SAT solver because it generates assignments in a deterministic lexicographic order. For a given function and a variable order, an SOP generated by using this method is unique and is independent of the input implementation of the function, the used SAT solver, and the encoding of the problem.

Canonical simulation vectors and canonical signatures

Moreover, assuming a function $f(x_1, \ldots, x_n)$, if the assignments of the d most left variables x_i, where $1 \leq i \leq d$ for some $d \leq n$, are fixed to some value, LEXSAT would generate an assignment that is lexicographically closest to the value defined when the d most left variables are assigned to the fixed values and the rest of the variables x_j, where $d+1 \leq j \leq n$ are assigned to 0.

Example 3.2.1. For the function $f(x_1, x_2, x_3, x_4)$ from Example 3.0.1, if we fix the most left variable x_1 to 1, then LEXSAT returns the assignment 1010 as lexicographically smallest because it is the satisfying assignment with the smallest integer value after the assignment 1000.

With this, LEXSAT enables generating canonical simulation vectors used to generate canonical signatures for Boolean functions using a SAT solver. Similarly, applications such as constraint solving [Yuan et al., 2004] and random assignment generation [Nadel, 2011] can benefit from LEXSAT, because it can derive the closest satisfying assignments for random valuations of inputs.

Canonicity enables caching results of intermediate computations

In general, because LEXSAT generates deterministic assignments, it enables canonicity in SAT-based applications with two important consequences: On the one hand, the result of computation depends only on the Boolean function and the user-specified variable order (and

is independent of the SAT solver and the problem representation, in particular, of the CNF generation algorithm). On the other hand, subproblems encountered during SAT solving can be cached in a way similar to how BDD-based applications cache the results of intermediate computations, resulting in runtime reduction. To this end, BDD-based applications maintain a hash table mapping BDD nodes into results of computation for these nodes. Similarly, a SAT-based application can use LEXSAT to compute a canonical representation of Boolean functions (such as the canonical SOP mentioned above). This canonical representation can be used as a hash key in a table of computed results, similarly to how BDD nodes are used as hash keys in BDD-based applications.

Approximate computing and bug characterisation

In addition, algorithms for approximate computing [Soeken et al., 2016b; Venkatesan et al., 2011] can use LEXSAT to compute the worst-case error by finding the lexicographically greatest solution for the difference between the approximate output and a correct reference version for all possible inputs. In formal verification, LEXSAT can analyse bugs that the SAT solver finds when solving verification instances. Suppose, for example, a satisfying assignment is found that indicates a mismatch between the specification and the implementation of a hardware design. LEXSAT can determine the lexicographically closest correct minterms before and after the buggy minterm. The difference between the two correct minterms outlines the region of the input space where the bug is present. When one bug is characterised in this way, a question can be asked: Are there other bugs before and after the given one in the lexicographical order? Repeatedly calling LEXSAT enables exploring the input space step-by-step, and understanding the distribution and the size of buggy regions, which could provide crucial information for debugging.

In summary, an appealing aspect of LEXSAT is that it enables canonicity in SAT-based applications, which leads to the same benefits BDD-based applications reap from the canonicity of BDDs that are unique for a given function and for a given variable order. Furthermore, there could be practically important applications of LEXSAT in verification, such as “canonical” random simulation based on evenly-distributed input patterns, or bug characterisation based on the exploration of input space performed by LEXSAT.

3.3 LEXSAT for SAT Algorithms

In this section, we present ideas for solving variations of the SAT problem by using LEXSAT or its concept: first, for solving the existing AllSAT, #SAT, and MAX-SAT, and then for the novel LEX-UNSAT problem. Future work can be focused on implementing and evaluating these ideas.

The AllSAT and #SAT problems

The AllSAT and #SAT problems are closely related. On the one hand, for a given CNF formula, the *all solutions satisfiability (AllSAT)* problem, which is also called *model enumeration*, generates partial satisfying assignments that form a logically equivalent *disjunctive normal form (DNF)* formula; a DNF formula F is a disjunction (OR, +) of cubes, $F = c_1 + \cdots + c_n$, where c_i for $1 \leq i \leq n$ represent cubes. The AllSAT problem has many diverse applications including data mining [Jabbour et al., 2013] and formal verification [Toda and Soh, 2016]. On the other hand, the *counting satisfiability (#SAT)* problem, which is also called *propositional model counting*, returns the total number of satisfying assignments. The #SAT became an important component in domains of artificial intelligence, planning, model checking, and hardware testing [Belle, 2016; Burchard et al., 2015].

Solving AllSAT and #SAT by using LEXSAT

To solve these two problems, we could identify regions in the Boolean space with successive satisfying assignments. First, we have to initialise two SAT solvers, one with the on-set and another with the off-set of the problem; for convenience we call them *on-set SAT solver* and *off-set SAT solver*, respectively. Then, we have to find, in an alternating fashion, LEXSAT assignments from the on-set and the off-set: By using the on-set SAT solver, we could first find the smallest LEXSAT assignment from the on-set, M_{on}, that comes after the last generated off-set assignment; by using the off-set SAT solver, we could then find the smallest LEXSAT assignment from the off-set, M_{off}, that comes after the last generated on-set assignment. The integer difference $M_{\text{off}} - M_{\text{on}}$ represents the number of on-set assignments in the Boolean space in the range $[M_{\text{on}}, M_{\text{off}})$. Thus, to solve #SAT, we should sum up the integer differences $M_{\text{off}} - M_{\text{on}}$ of all computed (M_{on}, M_{off}) pairs. Whereas, to solve AllSAT, we should generate DNF cubes that would represent the on-set assignments in the range $[M_{\text{on}}, M_{\text{off}})$.

Example 3.3.1. Assume a 4-input function $f(x_1, x_2, x_3, x_4)$ with the satisfying on-set assignments $\{0011, 0100, 0101, 1100, 1101, 1111\}$. First, with the on-set SAT solver, we can find the smallest on-set LEXSAT as-

signment that is greater than or equal to 0000, which is $M_{\text{on}} = 0011$. Next, using the off-set SAT solver, we can find the smallest off-set LEXSAT assignment greater than 0011, which is $M_{\text{off}} = 0110$. The integer difference between $M_{\text{off}} - M_{\text{on}}$ is 3, because there are three assignments in the range $[0011, 0110)$. Thus, for #SAT, we should add 3 to the final count of assignments. Whereas, for AllSAT, we should add to the DNF the disjoint cubes 0011 and 010− that cover the three assignments.

These LEXSAT-based methods can be particularly useful for problems that satisfy the following two conditions. First, obtaining the CNF description of the off-set should be easy. For a problem that represents a Boolean function, this can be performed by just negating the primary output. Unfortunately, if only the CNF is available, complementing it might be hard when the CNF is too large. Second, the on-set should be sparse but not fragmented. This method would be impractical if the onset is very fragmented, such as in the case of a multi-input XOR, when we would need to switch an exponential number of times between on-set and off-set.

The MAX-SAT problem and its variations

For a given propositional formula in CNF form, the *maximum satisfiability (MAX-SAT)* problem returns an assignment for the variables of the formula that satisfies the maximum number of clauses. MAX-SAT is usually executed for UNSAT problems, because in satisfiable problems all clauses are satisfied by the returned satisfying assignment. There are several variations of MAX-SAT, among which the most used are weighted MAX-SAT and partial MAX-SAT. For a given set of weights for the clauses, *weighted MAX-SAT* returns an assignment that maximises the sum of weights of the satisfied clauses. For a given division of the clauses into hard and soft clauses, the *partial MAX-SAT* returns an assignment that satisfies all hard clauses and the maximum number of soft clauses. Both the original problem and its variations are used in a wide range of domains, such as EDA [Chen et al., 2009; Le et al., 2013], data analysis and machine learning [Berg et al., 2015], and automotive configuration [Walter et al., 2013].

Solving MAX-SAT and its variations by using LEXSAT

LEXSAT can be used as yet another solution for the MAX-SAT problem, and notably, it can be easily extended to solve its variations. First, we have to add a variable that evaluates to 1 when the clause is satisfied. Next, these new clauses can be connected either with a sorting network or with an adder structure that would sum up that total number of satisfied clauses. A sorting network can be used only if all clauses have a

unit weight; whereas to solve the weighted MAX-SAT problem, for each clause, we can add a 2-to-1 multiplexer an input to the adder structure. This multiplexer outputs the weight of the clause when the clause is satisfied, or 0 otherwise. Finally, similarly to the approximate computing application from Section 3.2, we could then find an assignment for the variables of the original formula that maximises the output of the sorting network or the adder structure.

Novel algorithm that enables canonicity when using sets of assumptions for UNSAT

Finally, LEXSAT brings canonicity only for applications that rely on satisfiable calls. A solution is still required for applications that rely on UNSAT calls, because the received proof of unsatisfiability and the set of assumptions for UNSAT are non-canonical. By using the concept of LEXSAT, we can define and solve a new problem, called LEX-UNSAT, that brings canonicity whenever a set of assumptions for UNSAT is used.

When the problem is UNSAT, LEX-UNSAT returns a canonical set of assumptions for UNSAT

The *lexicographic unsatisfiability (LEX-UNSAT) problem* takes as input a propositional formula in CNF form, a given variable order, and a set of assumptions. If the problem is UNSAT, LEX-UNSAT returns a bit-stream in which each bit represents if the corresponding assumption belongs to the set of assumptions for UNSAT, and whose integer value is minimum (maximum) under the given variable order and assumptions. If the formula is satisfiable, LEX-UNSAT proves it satisfiable.

Comparison of LEXSAT and LEX-UNSAT

Compared to LEXSAT in which the returned bit-string represents an assignment for the input variables, LEX-UNSAT returns a bit-string in which each bit encodes if the corresponding assumption is used in the proof of unsatisfiability. An assumption represented with 1 is used and thus included in the set of assumptions for UNSAT, whereas an assumption represented with 0 is not included. Consequently, to minimise the returned bit-stream, LEXSAT that uses satisfying assignments, whereas LEX-UNSAT uses the set of assumptions for UNSAT.

Example 3.3.2. Assume a 5-input function $f(x_1, x_2, x_3, x_4, x_5)$ with the satisfying assignments for the inputs $\{00110, 00111, 01001, 11011\}$. If we call LEX-UNSAT for f with the assumptions $x_1 x_2 x_3 \bar{x}_4$, which would be encoded with the bit-string 11110, it would return the lexicographically smallest solution 00110 that proves that there is no satisfying assignment that satisfies the assumptions $x_3 \bar{x}_4$. Note that the set of assumptions for UNSAT $\bar{x}_4 \bar{x}_5$ that has globally the lexicographically smallest bit-string 00011 is not returned, as the returned solution also depends on the assumptions given as input.

3.4 Generating Lexicographic Satisfying Assignments

In this section, we first describe a simple and a binary search-based version of our algorithm for generation of LEXSAT assignments. Then, we describe several methods that improve their runtime when generating consecutive LEXSAT assignments.

3.4.1 Simple Version

The initial assignment is received as input

Instead of using a SAT solver to find the initial assignment, our algorithm receives as input an initial assignment $a_1 \dots a_n$ that, in this case, is smaller or equal to the next LEXSAT assignment. When generating consecutive LEXSAT assignments, this enables the search to start from the last generated LEXSAT assignment. For the first assignment or when generating non-consecutive assignments, for a function $f(x_1, \dots, x_n)$, the initial assignment is $a_i = 0$ for $1 \leq i \leq n$. Having this initial assignment, our algorithm iteratively verifies if the assignment of each variable can be fixed or if it should be increased. With this, it converts the initial assignment into the LEXSAT assignment that is returned as output.

Basic idea

A simple version of our algorithm fixes the assignments of the variables one by one. A pointer d, which is initially set to 1, gives the index of the first non-fixed variable whose assignment should be fixed, whereas for the previous variables the assignments $x_i = a_i$, for $1 \leq i < d$, are already fixed. To fix the assignments, a SAT solver is called iteratively with the assumptions $x_i = a_i$, for $1 \leq i \leq d$. If the problem is SAT, then there is a satisfying assignment that starts with $a_1 \dots a_d$ and d is incremented. Otherwise, if there is no satisfying assignment that starts with $a_1 \dots a_d$, the problem is UNSAT. In this case, if $a_d = 0$, we set $a_d = 1$, set $a_i = 0$ for $d < i \leq n$ to keep the assignment the smallest possible for the future iterations, and perform another iteration. But, if the problem is UNSAT when $a_d = 1$, then there is no satisfying assignment both when $a_d = 0$ and $a_d = 1$, hence the algorithm returns UNSAT. Once $d > n$, the assignments for all variables are fixed and $a_1 \dots a_n$ is returned as a LEXSAT assignment.

Example 3.4.1. To generate the first LEXSAT assignment for a function $f(x_1, x_2, x_3, x_4, x_5)$, the received initial assignment is 00000. Initially, $d = 1$ and the first SAT call assumes $x_1 = 0$. If the problem is SAT, then d

is incremented to $d = 2$, and in the next iteration the SAT call assumes $x_1 = 0$ and $x_2 = 0$. Otherwise, if the problem is UNSAT, we flip $a_1 = 1$, and iterate with the assumption $x_1 = 1$. This time, if we receive SAT, we increment d, and in the next iteration the SAT call assumes $x_1 = 1$ and $x_2 = 0$. But, if we receive UNSAT again, it means that there is no assignment both with $x_1 = 0$ and $x_1 = 1$, and thus we return UNSAT.

Improving performance by learning from satisfying assignments

Similarly to the algorithm by Knuth [2015] described in Section 3.1, when the SAT solver returns a satisfying assignment, we can learn some variable assignments from it. Thus, we always save the last satisfying assignment, and use it as follows. First, same as before, if the first variable assigned to 1 after d is on position $d+t$, where $1 \leq t \leq n-d$, then we can learn and fix to 0 the $t-1$ variables between d and $d+t$. Moreover, in our case, the potential assignment $a_1 \dots a_n$ is the lexicographically smallest assignment that might be satisfying. Thus, if the potential assignment for a variable x_i is $a_i = 1$, then we cannot flip it to 0 in order to minimise the assignment as in the algorithm by Knuth. This enables us to learn from the SAT solver and fix all assignments up to the first variable for which the potential assignment and the assignment returned by the SAT solver differ. Assume that the last satisfying assignment returned by the solver is $v_1 \dots v_n$. Instead of incrementing d by 1, we can set it to the index i, such that $a_j = v_j$ for $1 \leq j < i$ and $a_i \neq v_i$. Finally, same as Knuth's algorithm, for a given literal x_d, where $1 < d \leq n$, with $v_d = 1$, if we get UNSAT when assuming $x_d = 0$, we can immediately fix x_d to 1, as this value is confirmed by the last satisfying assignment.

Example 3.4.2. For a function $f(x_1, \dots, x_6)$, assume that 101000 is received as an initial assignment. When the SAT solver is called with the assumption $x_1 = 1$, it returns a satisfying assignment 101101, which is saved as a last satisfying assignment. Besides fixing $x_1 = 1$, from this assignment, we can learn and fix $x_2 = 0$ and $x_3 = 1$, because their initial assignments are confirmed by the last satisfying assignment. The variable x_4 is the left-most variable for which the assignments differ and might be flipped to 0, so for the next iteration we set $d = 4$ and call the SAT solver with the assumptions $x_1 = 1$, $x_2 = 0$, $x_3 = 1$ and $x_4 = 0$. If the problem is SAT, we fix x_4 to 0 and update the last satisfying assignment. But, if the problem is UNSAT, from the last satisfying assignment 101101, we already know that the problem is satisfiable when $x_4 = 1$, we can additionally fix $x_5 = 0$, and set $d = 6$ for the next iteration.

3.4.2 Binary Search-Based Version

Speculative version based on binary search

To further enhance the simple version of our algorithm, instead of fixing the assignments of variables one by one, we propose to set the pointer d using binary search. Two additional pointers l and r show the first and last variable with non-fixed assignments, respectively, and initially are set $l = 1$ and $r = n$. Then, d is set to the middle variable in the array of variables bounded by x_l and x_r. This assumes the assignments of the left half of the variables x_i, where $1 \leq i \leq d$, in the first iteration. Later, whenever the SAT solver returns SAT, it confirms that a satisfying assignment that starts with $a_1 \dots a_d$ exists. As shown in Section 3.4.1, from the returned satisfying assignment, we can confirm and fix t additional assignments from the potential assignment, where $0 < t < n - d$. After this step, the assignments for the variables x_i, where $1 \leq i \leq d + t$ are fixed. For the next iteration, we set $l = d + t + 1$ and $r = n$ to assume the assignments for the non-fixed variables in the right half. Otherwise, if the problem is UNSAT, if $a_d = 0$, then we proceed as in the simple version of the algorithm: we set $a_d = 1$, set $a_i = 0$ for $d < i \leq n$ for the future iterations, and perform another iteration; whereas, if $a_d = 1$, for the next iteration $r = d - 1$ to assume fewer non-fixed variables.

Example 3.4.3. To generate the first LEXSAT assignment for a function $f(x_1, x_2, x_3, x_4, x_5, x_6)$, the initial assignment 000000 is received as input. Initially, $l = 1$, $r = 6$ and $d = 3$. Thus, the first SAT call would assume $x_1 = 0$, $x_2 = 0$, and $x_3 = 0$. If the problem is SAT and the satisfying assignment 000010 is returned, then the assignment $x_4 = 0$ is learned as it is the same in the initial assignment, and the values of the pointers are updated to $l = 5$, $r = 6$ and $d = 5$ for the next iteration. Otherwise, if it is UNSAT, we would first try the assumptions $x_1 = 0$, $x_2 = 0$, and $x_3 = 1$. This time, if we receive SAT we would proceed same as before; whereas, if we receive UNSAT again, for the next iteration, we would update the values of the pointers to $l = 1$, $r = 2$ and $d = 1$ to assume fewer variables.

3.4.3 Runtime Improvement for Consecutive LEXSAT Assignments

Generation of consecutive LEXSAT assignments

Applications such as the SAT-based generation of canonical SOPs, which is presented in Chapter 4, generate consecutive satisfying assignments in lexicographic order. To enable the generation of new satisfying assignments, each generated assignment is added to the SAT solver as a

blocking clause, which is an additional clause that blocks known solutions of the SAT problem.

Example 3.4.4. For the function $f(x_1, x_2, x_3, x_4)$ from Example 3.0.1, the first LEXSAT call returns the assignment 0001. If we add this assignment as a blocking clause to the SAT solver, with the next LEXSAT call the assignment 0101 is generated because it is the lexicographically smallest satisfying assignment that is not blocked.

Methods for runtime improvement

For these types of algorithms, we present three methods that improve the runtime of the newly proposed algorithms. These methods benefit from (1) the lexicographic properties of the assignments and (2) the fact that, after the first LEXSAT call, the received initial assignment is the last generated LEXSAT assignment.

Fixing leading 1s

When generating consecutive LEXSAT assignments, after some time, assignments that start with one or more consecutive 1s are generated. Generating a lexicographically smallest satisfying assignment that starts with one or more consecutive 1s implies that all unblocked satisfying assignments are greater than the generated one, hence they also start with the same number of 1s. Thus, when generating a LEXSAT assignment, assume that $a_i = 1$ for $1 \leq i \leq t$, for some $t \leq n$ (i.e., the received initial assignment starts with t consecutive 1s). Then, we can fix these t assignments for the corresponding variables x_i, and the initial value of l (or of d in the simple version) is set to $t+1$ to point the first variable that is assigned 0.

Example 3.4.5. For a function $f(x_1, x_2, x_3, x_4, x_5)$, assume that the last generated LEXSAT assignment is 11010 and it is received as an initial assignment. As the next LEXSAT assignment has to be greater than the last generated assignment, we know that it also starts with 11. Hence, we can skip assuming assignments for x_1 and x_2, and directly fix them to 1. Initially, l is set to 3, r to 5, d is computed to be 4, and thus, the first SAT call would be with the assumptions $x_1 = 1$, $x_2 = 1$, $x_3 = 0$, and $x_4 = 1$.

Correcting the initial assignment

When generating consecutive LEXSAT assignments, for the first LEXSAT assignment, the initial assignment received as input assigns all variables to 0. Afterwards, for the following LEXSAT assignments, the initial assignment is equal to the last generated LEXSAT assignment. But, the first unblocked assignment is the one whose integer value is one unit

greater than the last LEXSAT assignment. Thus, assuming that the last LEXSAT assignment ends with t 1s, for some $t \leq n$, i.e., $a_{n-i} = 1$ for $0 \leq i < t$, we flip the most right 1s by setting $a_{n-i} = 0$ and the first 0 from the right by setting $a_{n-t} = 1$.

Example 3.4.6. For a function $f(x_1, x_2, x_3, x_4, x_5)$, assume that the assignment 11011 is generated with the previous LEXSAT call and received as an initial assignment. As the next lexicographical assignment has to be greater than the last generated, the first possible satisfying assignment is 11100. Thus, we flip the 1s and the first 0 starting from the right to get the potential assignment 11100.

Profiling the success of the first SAT calls

For the LEXSAT algorithm, we consider satisfiable SAT calls as successful because they confirm the assumed assignments, and unsatisfiable SAT calls are considered unsuccessful. Furthermore, we propose to profile the success of the first SAT call from the LEXSAT algorithm and use this profile to alter the percentage of assumed assignments in the first SAT calls in the subsequent invocation of the LEXSAT algorithm based on binary search. This method does not apply to the simple version of the algorithm.

The binary search-based version always sets the pointer d to point the middle variable of the array of variables bounded by x_l and x_r. Thus, with the first SAT call, we always assume the non-fixed assignments for the first 50% of the variables between x_l and x_r. In the next iterations, with every satisfiable SAT call, we increase the number of assumptions and add 50% more of the right subarray. With every unsatisfiable SAT call, we decrease the number of assumptions and the next time we use only 50% of the assignments of the left subarray; for example, assuming 75% of the assignments in the first SAT call is equivalent to having two consecutive iterations with successful SAT calls.

To profile and alter the percentage of assumed assignments in the first SAT call, we keep a variable w that tells us how many iterations to perform at once and in which direction we should perform them. We iterate $|w|$ times to decrease or increase the percentage when $w < 0$ or $w > 0$, respectively. Initially, $w = 0$, which means that we should assume 50% of the assignments. If the first SAT call is satisfiable, we increase w for 1 when $w \geq 0$ or we set $w = 1$ when $w < 0$. If the first SAT call is unsatisfiable, we decrease w for 1 when $w \leq 0$ or we set $w = 0$ when $w > 0$. Figure 3.1 shows how the percentage of assumed variables for

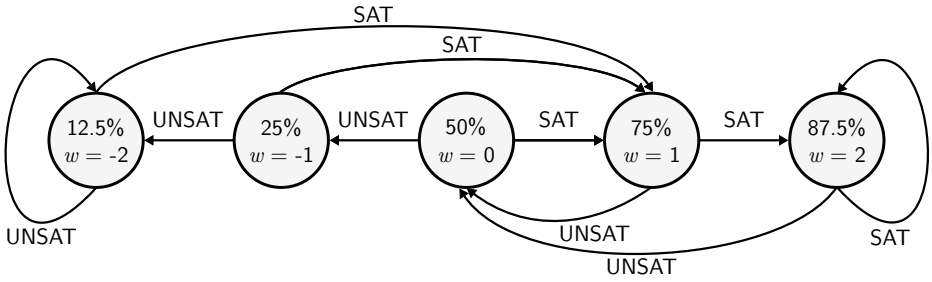


Figure 3.1 – Profiling the success of the first SAT calls. The percentage of assumed variables for the first SAT call of LEXSAT is changed depending on the success of the previous first SAT calls. In this case, at most three iterations of binary search are performed at once.

the first SAT call and the value of w changes with the success of the first SAT calls. In this example, at most three iterations of binary search are performed at once.

Example 3.4.7. For a function $f(x_1,\ldots,x_{10})$, assume that the assignment 0000110000 is received as an initial assignment and $w = 0$. As $l = 1$, $r = 10$ and $w = 0$, for the first SAT call d is computed as $d = \lfloor(1+10)\cdot 0.5\rfloor = 5$. Thus, we assume $x_1 = 0$, $x_2 = 0$, $x_3 = 0$, $x_4 = 0$ and $x_5 = 1$. If this call is satisfiable, then w is set to 1 for the next LEXSAT assignment. Assume that with the following SAT calls the LEXSAT assignment 0000110001 is generated. Then, when generating the next LEXSAT assignment, for the first SAT call, $d = \lfloor(1+10)\cdot 0.75\rfloor = 8$ because $w = 1$, so instead of assuming the initial assignments only for the first five inputs as before, we assume the assignments for the first eight inputs. For the remaining SAT calls of the current LEXSAT assignment, we always use the regular binary search algorithm that always assumes 50% of the assignments.

3.5 Experimental Results

Algorithms implemented in ABC

In this section, for convenience the algorithm from Knuth [2015] is called `KLEX` (Section 3.1), and the simple and binary search-based versions of our algorithm are called `SIMPLE` and `BINARY`, respectively (Section 3.4). We implemented in ABC [ABC] the three algorithms `KLEX`, `SIMPLE`, and `BINARY`, as well as the methods for improving the runtime from Section 3.4.3. ABC features an integrated incremental SAT solver derived from an early version of MiniSAT [Eén and Sörensson, 2003].

Also, this SAT solver supports interfaces for pushing and popping of assumptions that we use in our implementations.

Benchmarks

To evaluate the runtime of the algorithms and the speedup achieved from the additional methods, we use the set of large MCNC benchmarks and a set of logic tables from the instruction decoder unit [BenchIBM], which we denote with LT-DEC. The names of the LT-DEC benchmarks are given in the form "[N_{PI}].[N_{PO}]", where N_{PI} is the number of primary inputs and N_{PO} is the number of primary outputs. For a given benchmark, each algorithm generates a user specified number of consecutive LEXSAT assignments for each *combinational output* that is each primary output and each latch input. However, to avoid repeatedly calling the procedure for output functions with isomorphic circuit structure, we divide the outputs into equivalence classes. An *equivalence class* contains outputs that implement an identical function expressed over different inputs. Hence, for each benchmark, we actually generate LEXSAT assignments only for the representative of each class.

Correctness checking

For a given function and a variable order, the LEXSAT assignments are deterministic and must be generated in the same order when generating consecutive LEXSAT assignments. The correctness of our algorithms is validated by generating assignments with each algorithm, and comparing them to ensure that all algorithms generate the same assignments in the same order. For generating a given number of LEXSAT assignments, the number of SAT calls depends on how often the algorithm calls the SAT-solving procedure.

Next, we compare the runtime of `KLEX` and the two versions, `SIMPLE` and `BINARY`, of our algorithm that are enhanced with the methods described in Section 3.4.3. We evaluate the three algorithms for both generation of non-consecutive and consecutive LEXSAT assignments. Then, we show the influence of the variable order on the runtime. Afterwards, we evaluate the speedup achieved by each of the additional methods.

3.5.1 Runtime Comparison

Generation of non-consecutive LEXSAT assignments

Applications such as the NPN classification [Soeken et al., 2016a] require multiple LEXSAT assignments, but they are not in a consecutive order or they are for different functions. Therefore, we evaluate the runtime and number of SAT calls required by each algorithm for generating a

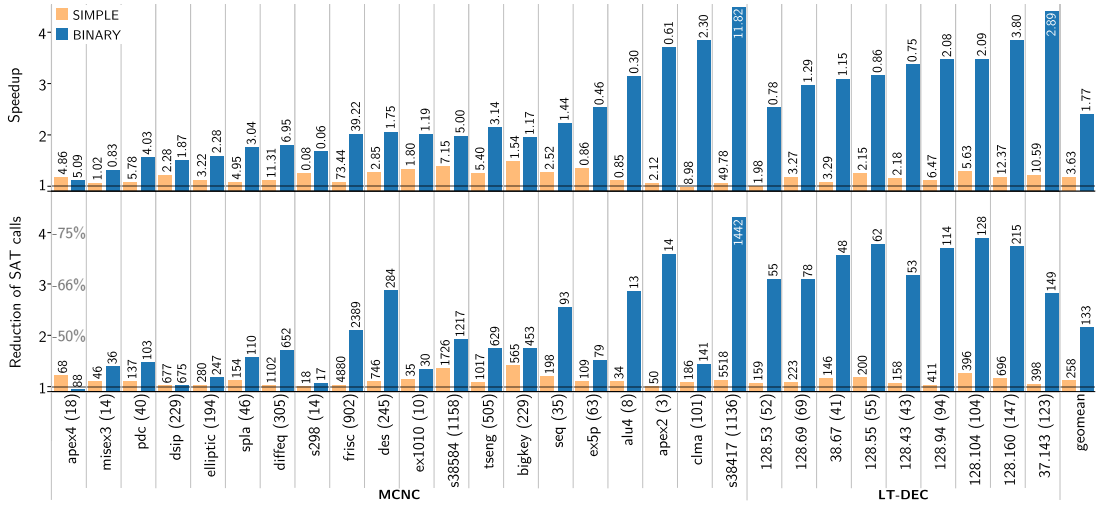


Figure 3.2 – Performance of the LEXSAT algorithms when generating 1000 consecutive assignments. We show the speedup and reduction of the number of SAT calls achieved by our algorithms `SIMPLE` and `BINARY` compared to the `KLEX` algorithm when generating a single LEXSAT assignment per combinational output. Next to each bar is the actual runtime (in milliseconds) and the number of SAT calls, respectively. Next to the name of the benchmark, we give the number of LEXSAT calls in brackets.

single LEXSAT assignment. For each benchmark, the smallest LEXSAT assignment is generated per combinational output. As the algorithms generate these assignments in few milliseconds, to compare them precisely, we generate each LEXSAT assignment 1000 times, and then divide the total runtime by 1000. As Figure 3.2 shows, both versions `SIMPLE` and `BINARY` perform better than `KLEX` for almost all benchmarks. Because the algorithmic steps of `SIMPLE` are very similar to those of `KLEX` when generating a single assignment, `SIMPLE` makes only 9.7% less calls to the SAT solver, and thus is only 14.7% faster than `KLEX`. On the contrary, assuming more assignments at once with `BINARY` leads to about 2x less SAT calls and 2x faster runtime than `SIMPLE`. Finally, `BINARY` is 2.4x faster than KLEX and makes 2.1x less SAT calls.

Generation of consecutive LEXSAT assignments

Some applications, such as the LEXSAT-based generation of canonical SOPs, which is presented in Chapter 4, require consecutive LEXSAT assignments. In this case, the methods described in Section 3.4.3 also contribute to reducing the runtime of `SIMPLE` and `BINARY`. For this experiment, we generate at most 1000 consecutive LEXSAT assignments for each combinational output. For each output we perform the experiment 5 times, thus the presented results represent the average over 5

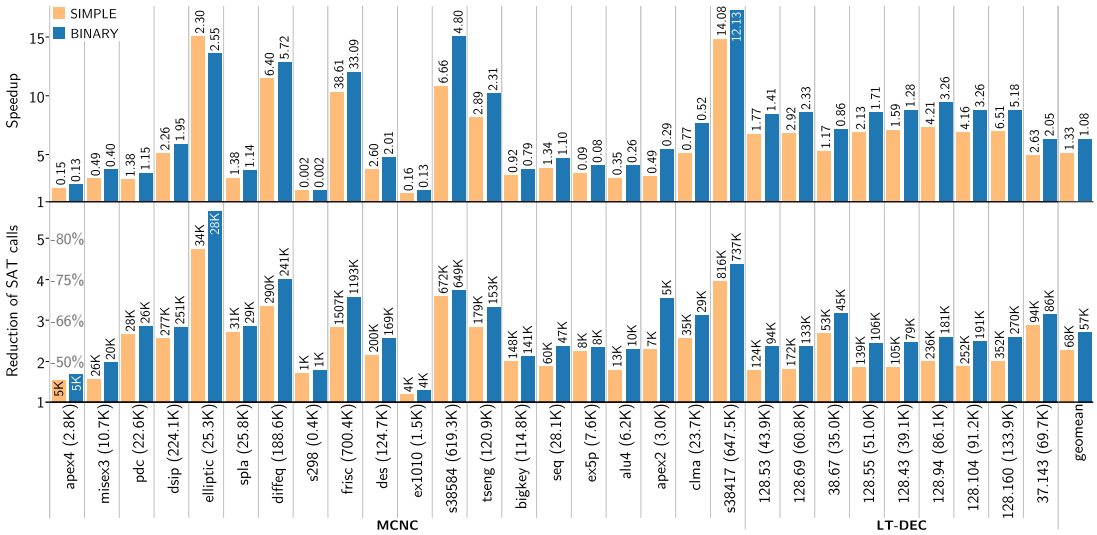


Figure 3.3 – Performance of the LEXSAT algorithms when generating 1000 consecutive assignments. We show the speedup and reduction of the number of SAT calls achieved by our algorithms `SIMPLE` and `BINARY` compared to `KLEX` when generating 1000 consecutive LEXSAT assignments per combinational output. Next to each bar is the actual runtime (in seconds) and the number of SAT calls (in thousands), respectively. Next to the name of the benchmark, in brackets, is the number of LEXSAT calls (in thousands).

runs. As Figure 3.3 shows, both `SIMPLE` and `BINARY` outperform `KLEX`: `SIMPLE` makes 2.3x less SAT calls on average, which reduces runtime 5.1x; whereas, `BINARY` makes 2.7x less SAT calls on average, which reduces runtime 6.3x. Regarding the two proposed versions of our algorithm, on average, `BINARY` has 16.1% less SAT calls that contribute to 18.9% better runtime than `SIMPLE`.

Thus, `BINARY` has the best performance both when generating non-consecutive and consecutive LEXSAT assignments.

3.5.2 Influence of the Variable Order on the Runtime

The results and the runtime might differ if the variable order is changed

The results of LEXSAT depend on the used variable order. As illustrated by Example 3.5.1, the smallest LEXSAT assignment might be different for different variable orders.

Example 3.5.1. For the function $f(x_1, x_2, x_3, x_4)$ from Example 3.0.1, the smallest LEXSAT assignment for the original variable order (x_1, x_2, x_3, x_4) is 0001. In contrast, for the reverse variable order (x_4, x_3, x_2, x_1), the

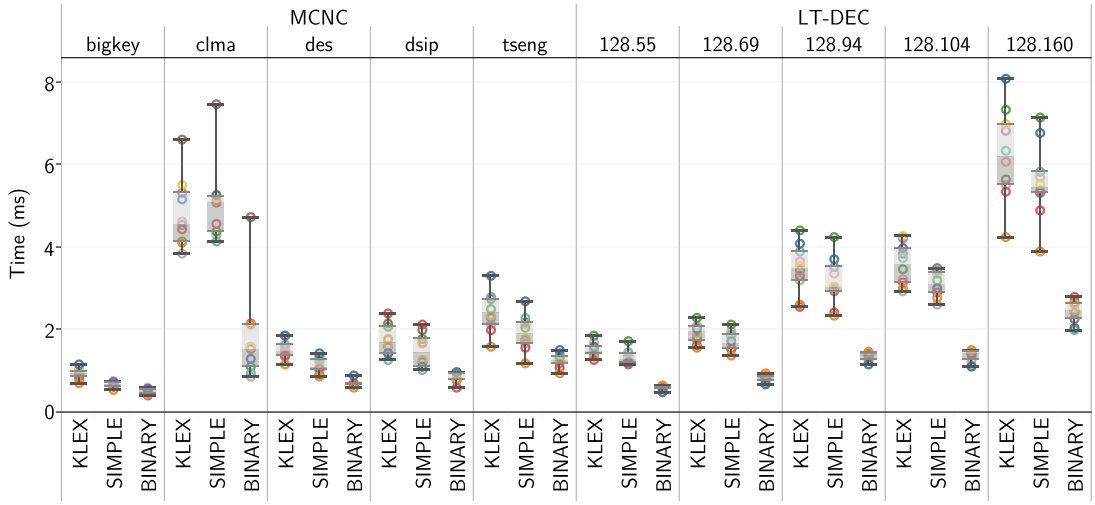


Figure 3.4 – Influence of the variable order on the runtime of the LEXSAT algorithms when generating a single assignment. For the ten selected benchmarks, we show the runtime of `KLEX`, `SIMPLE`, and `BINARY` for ten different variable orders. Each circle represents the runtime (in milliseconds) for one variable order, and the box-and-whisker plots show the median and the quartiles.

smallest LEXSAT assignment is 0101, which is equivalent to 1010 when considering the original variable order.

Although some applications have to use one specific variable order, other applications are more flexible and can change the variable order to improve the runtime. In particular, from the described LEXSAT applications in Section 3.2 and Section 3.3, NPN classification, correction of cell placement, approximate computing, and MAX-SAT have a restriction in the variable order and cannot change it; whereas, the applications for generation of canonical simulation vectors and canonical signatures, bug characterisation, AllSAT, and #SAT can work with different variable orders.

Experimental setup

In this section, we evaluate the influence of the variable order on the runtime when generating both non-consecutive and consecutive LEXSAT assignments. Similarly to the experimental setup from Section 3.5.1, for non-consecutive LEXSAT assignments, we generate a single assignment for each combinational output and the reported results are average over 1000 runs; whereas, for consecutive LEXSAT assignments, we generate 1000 assignment for each combinational output and the reported results are average over 5 runs. For each benchmark, we use ten different variable orders: the pre-defined variable order from the benchmark file

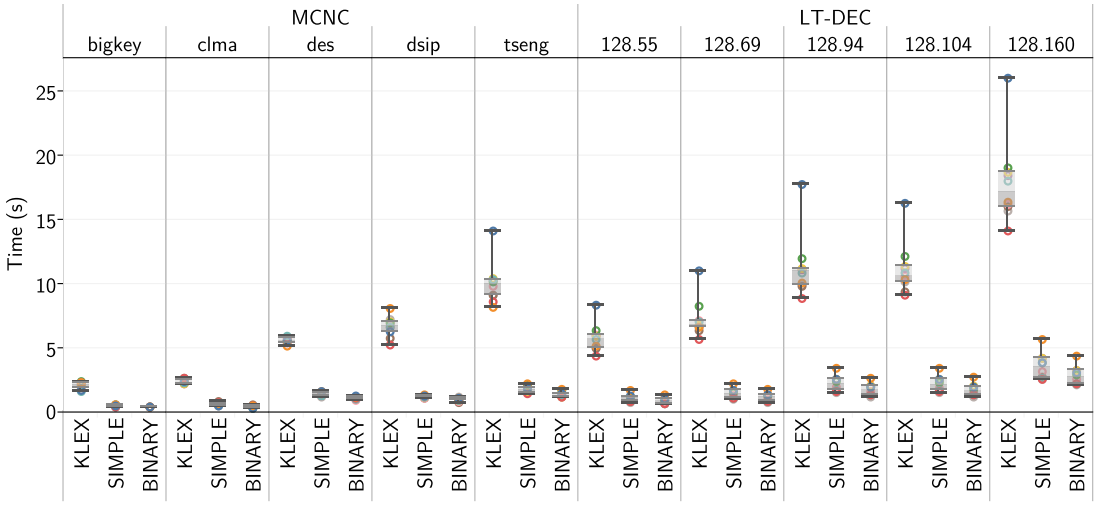


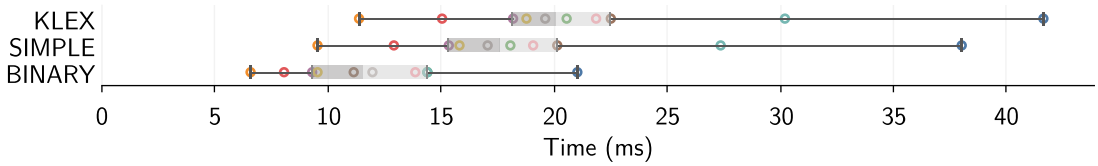
Figure 3.5 – Influence of the variable order on the runtime of the LEXSAT algorithms when generating 1000 consecutive assignments. For the ten selected benchmarks, we show the runtime of `KLEX`, `SIMPLE`, and `BINARY` for ten different variable orders. Each circle represents the runtime (in seconds) for one variable order, and the box-and-whisker plots show the median and the quartiles.

and four randomised variable orders obtained by swapping variables, as well as their five reversed versions. Note that the runtime results presented in Figures 3.4, 3.5 and 3.6 do not match with the absolute runtime presented in Figures 3.2, 3.3, 3.7 and 3.8 because the experiments are executed on different machines with different configuration and performance.

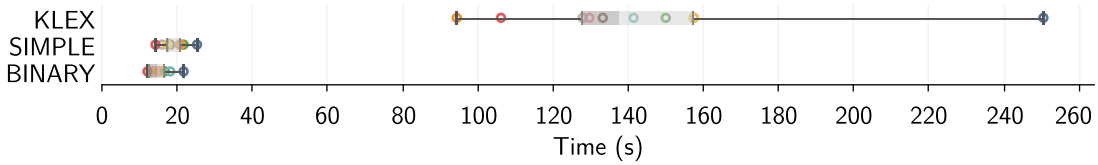
Results for ten different variable orders

For most benchmarks, the variable order does not have a significant influence on the runtime. Figure 3.4 and Figure 3.5 show the results for ten benchmarks when generating non-consecutive and consecutive LEXSAT assignments, respectively. From each set, we show the five benchmarks with the highest number of PIs; if two benchmarks have equal number of PIs, we show the one with the highest number of POs. From all benchmarks, we observe the largest difference in runtime for the benchmark frisc from the MCNC set, which can be seen in Figure 3.6. These results reveal that, from the three algorithms, `KLEX` is the most sensitive to the variable order and `BINARY` is the least sensitive, which is yet another advantage of the algorithm `BINARY`.

Note that the reported results in Section 3.5.1 are generated by using the pre-defined variable order from the benchmark file.



(a) Runtime when generating a single LEXSAT assignment.



(b) Runtime when generating 1000 consecutive LEXSAT assignments.

Figure 3.6 – Runtime variation for the benchmark frisc from the MCNC set for different variable orders. Each circle represents the runtime of the given LEXSAT algorithm for one variable order, and the box-and-whisker plots show the median and the quartiles.

3.5.3 Evaluation of the Methods for Runtime Improvement

The three methods for runtime improvement

Section 3.4.3 presented the following methods for runtime improvement when generating consecutive assignments.

1. Fixing leading 1s.
2. Correcting the initial assignment.
3. Profiling the success of the first SAT calls.

Evaluation by generating the complete truth table of small benchmarks

As the method for fixing the leading 1s affects the runtime only when generating assignments in which the most significant bits are assigned to 1, we evaluate the methods by generating the complete truth table (i.e., generating all assignments for which the function evaluates to 1) for a subset of the MCNC benchmarks. The selected benchmarks have at most 16 combinational inputs, which means that, for each combinational output, we can have at most 65536 minterms when the function is 1. As in Section 3.5.1, the presented results represent the average over 5 runs. Figure 3.7 shows the runtime and number of SAT calls for four of the selected benchmarks. First, it shows the results when the algorithms `SIMPLE` (S) and `BINARY` (B) are used without the additional methods. We can see that fixing the leading 1s (S+1, B+1) decreases the runtime moderately. On the contrary, if we additionally correct the initial assignment (S+1+2, B+1+2) then the runtime decreases by 32%, on average. Finally, for `BINARY`, although the method for profiling the

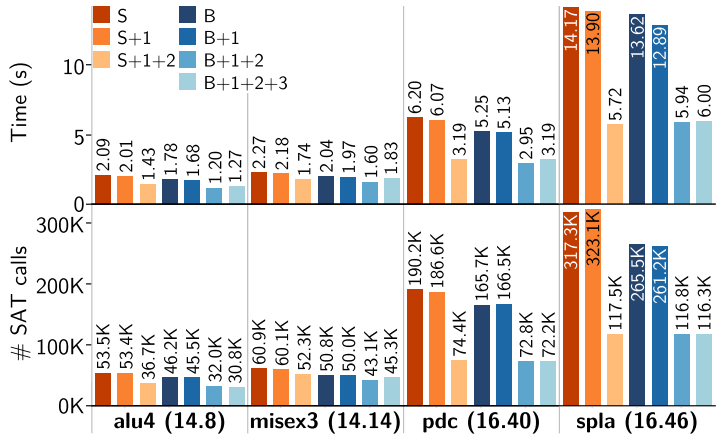


Figure 3.7 – Effect of the methods for runtime improvement on small benchmarks. The runtime and number of SAT calls of `SIMPLE` (S) and `BINARY` (B) are shown when different methods for improving the runtime are used for 4 benchmarks from the MCNC set. Next to the name of the benchmark, we give the number of combinational inputs and outputs, respectively.

success of the first SAT calls in general decreases the number of SAT calls, for functions with small number of inputs it slightly increases the runtime.

Evaluation by generating 1000 minterms for large benchmarks

We have observed, however, a reduction of runtime for benchmarks with a large number of combinational inputs. Figure 3.8 shows the runtime and number of SAT calls required to generate 1000 minterms for a single output of 4 large MCNC benchmarks. The considered outputs have more than 70 combinational inputs. In this case, the method for fixing leading 1s does not affect the number of SAT calls because the most significant bits of all generated assignments are 0s.

Note that in Section 3.5.1 the results for `SIMPLE` and `BINARY` are obtained when all methods are used (i.e., with S+1+2 and B+1+2+3, respectively).

3.6 On Integrating the LEXSAT Algorithms in a SAT Solver

Two options for implementing LEXSAT

The algorithms presented and evaluated in this chapter repeatedly use the SAT solver. Another option is to modify the SAT solver such that it generates LEXSAT assignments. For convenience, we call them `OUTSAT`

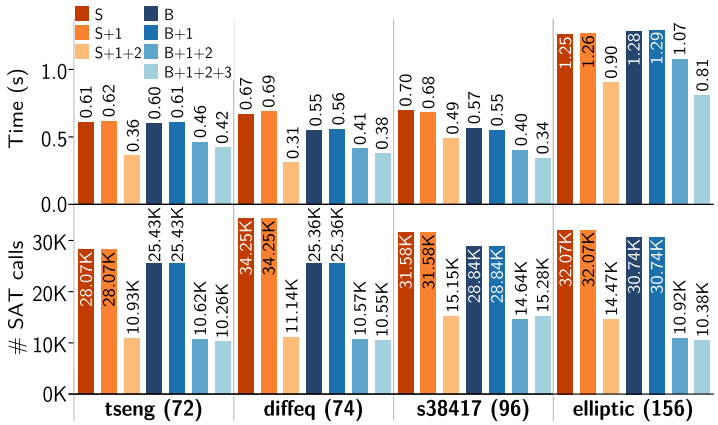


Figure 3.8 – Effect of the methods for runtime improvement on large benchmarks. The runtime and number of SAT calls of SIMPLE (S) and BINARY (B) are shown when different methods for runtime improvement are used for one of the outputs from 4 benchmarks from the MCNC set. Next to the name of the benchmark, we give its number of combinational inputs.

and INSAT, respectively. Knuth [2015, Ex. 7.2.2.2-275] suggests an INSAT implementation of KLEX. Nadel and Ryvchin [2016] show that an INSAT algorithm is faster than an OUTSAT implementation of the KLEX algorithm, but unlike the OUTSAT implementation, it is not scalable for difficult instances. In this section, we discuss the difference in these two implementation options.

Characteristics of an INSAT implementation

Generally, in an INSAT implementation, the SAT solver performs decisions on the input variables in the order and with the values given by the LEXSAT algorithm, whereas for the other variables decisions can be performed in any order. With this solution, to generate LEXSAT assignments for a function, a single SAT solver instance is created and, for each LEXSAT assignment, the procedure for SAT solving is called only once, with a given order for the input variables. Note that the concepts of the algorithms SIMPLE and BINARY can also be used to determine the order of issuing decisions and the values for the input variables.

Characteristics of an OUTSAT implementation

In our OUTSAT implementations, incremental SAT solving also enables generating multiple LEXSAT assignments of a function by using only a single SAT solver instance. Moreover, for each LEXSAT assignment, the interfaces for pushing and popping assumptions, which we suggest to use, preserve the internal state of the solver between consecutive invocations of the SAT-solving procedure. With this, on a higher level,

we mimic the solution based on modifying the SAT solver. With such implementation, and by using the algorithm BINARY, we expect our OUTSAT implementation to be as fast as an INSAT implementation, but confirming this experimentally is left for future work.

Flexibility of an OUTSAT implementation

Moreover, assume a function with n inputs for which the assignments of the first d inputs are already fixed, from some $1 \leq d \leq n$. In the OUTSAT implementation, the SAT-solving procedure can and do change the order of decisions for the least significant $n-d-1$ variables whose value is not yet fixed, when running the query to fix the value of the variable $d+1$. However, the INSAT implementation always makes the same decisions in the same order, and cannot change the order, even if this would lead to faster UNSAT calls during LEXSAT solving. Hence, for difficult instances, such as functions with large number of variables when different variable orders affect the efficiency of the SAT-solving procedure, as well as when all calls are not satisfiable, an OUTSAT implementation is more scalable than an INSAT implementation.

3.7 Conclusion

Key insights

In this chapter, we have presented a novel variation of the Boolean satisfiability problem, called LEXSAT. In addition to determining the status of a problem (satisfiable or unsatisfiable), LEXSAT returns satisfying assignments that are minimum (maximum) considering a given variable order. We demonstrate that LEXSAT enable the development of SAT-based algorithms that share desirable properties with BDDs but are less likely to suffer from the scalability problems that beset BDD-based computations in many EDA applications. In particular, LEXSAT can achieve *canonicity* of the computed results: For a given Boolean function and a given variable order, the result is deterministic and independent of the SAT solver and the CNF generation algorithm. In the next chapter, we demonstrate how this property of LEXSAT can be used for generation of canonical SOPs.

We have also proposed a fast binary search-based algorithm for the generation of LEXSAT assignments: it is 2.4 times faster than the state-of-the-art LEXSAT algorithm. Furthermore, it proposes several improvements to the LEXSAT algorithms for situations when it is applied iteratively and the resulting satisfying assignments are monotonically

increasing. For such use, our proposed algorithm enhanced with the new features is 6.3 times faster than the state-of-the-art LEXSAT algorithm. Finally, to improve the performance without modifying the SAT solver, we have proposed a way of using incremental SAT solving with pushing and popping of assumptions.

Future applications of LEXSAT

We expect LEXSAT to be applied in many EDA applications, such as those presented in Section 3.2. Future work on LEXSAT could also focus on exploring several other promising applications: SAT-based constraint simulation, SAT-based factoring, SAT-based exclusive sum-of-product minimisation, etc. Finally, as discussed in Section 3.3, other variations of the SAT problem can be implemented by using LEXSAT, and future work could implement and evaluate them.

4 Progressive Generation of Canonical Irredundant Sums of Products

Using SOPs in a variety of applications

Minimisation of the two-level *sum of products (SOP)* representation is well-studied due to the wide use of SOPs. In the past, SOPs were principally used for mapping into *programmable logic arrays (PLAs)*; now SOPs are supported in many tools for logic optimisation and are used for multi-level logic synthesis [Brayton et al., 1984; Rajski and Vasudevamurthy, 1992], delay optimisation [Mishchenko et al., 2011a, 2017], and test generation [Ghosh et al., 1991], but they are also used for fuzzy modelling [Gobi and Pedrycz, 2007], data compression [Amarù et al., 2014b], photonic design automation [Condrat et al., 2011] and in other areas.

SOPs for delay optimisation

These publications show that, contrary to popular belief, research in SOP minimisation and its applications are not outdated. For example, a recent work uses SOPs for delay optimisation in technology independent synthesis and technology mapping [Mishchenko et al., 2011a]. In this work, improved quality is achieved by enumerating different SOPs of the local functions of the nodes, factoring them, and finding circuit structures balanced for delay.

SOPs for global circuit restructuring

Another important application of SOP minimisation, which is used as a case-study in this chapter, is *global circuit restructuring*. If a multi-level circuit structure for a Boolean function is not available, or if the circuit structure is of a poor quality, then a new circuit structure with desirable properties, such as low area, short delay, good testability or improved implicativity (if the circuit represents constraints in a SAT

This chapter is based on the work of a book chapter that will appear in a book published by Springer [Petkovska et al., 2017].

solver) should be derived. The best known and widely used method for global circuit restructuring is computing SOPs of the output functions in terms of inputs, factoring the multi-output SOPs, and deriving a new circuit structure from the shared factored form. The main drawback of this method is the lack of scalability of the algorithm for SOP generation and minimisation.

Algorithms for SOP generation and minimisation

Starting with the Quine-McCluskey algorithm [McCluskey, 1956; Quine, 1952], many algorithms and heuristics for SOP generation and minimisation have been developed. Prior research falls into two broad categories: BDD-based algorithms and ESPRESSO-style algorithms.

BDD-based algorithms for generation and minimisation of SOPs

In order to generate an SOP for a given Boolean function, techniques based on BDDs, such as those of Minato-Moreale [Minato, 1992] and SCHERZO [Coudert, 1994; Coudert et al., 1993b], first build a BDD or a ZDD, then minimise the BDD/ZDD size by using some heuristic approach to obtain a smaller SOP, and finally convert the BDD/ZDD to an SOP. If building a BDD is feasible, then an SOP, even a suboptimal one, can be generated. However, as previously explained, using BDDs is often impractical due to the BDD memory explosion problem. An additional drawback is that BDDs are incompatible with incremental applications as they require building a BDD for the complete circuit before converting it to an SOP. Despite these issues, to our knowledge, the BDD-based method for SOP generation and minimisation is used in most of the industrial tools, therefore scalability improvements of it are highly desirable.

Algorithms for minimisation of SOPs based on ESPRESSO

The ESPRESSO-style algorithms are inspired by the first version of ESPRESSO [Brayton et al., 1984]. For example, the logic minimiser ESPRESSO-MV [Rudell and Sangiovanni-Vincentelli, 1987] is a faster and more efficient version of ESPRESSO. But, although these techniques avoid the memory explosion problem inherent in the use of BDDs, they still incur impractical runtimes for large Boolean functions and only minimise existing SOPs.

Existing SAT-based solutions

Although SAT solvers were proposed as an alternative for many EDA applications, to the best of our knowledge, there is still no complete SAT-based method for SOP generation similar to the *irredundant sum of products (ISOP)* algorithm for incompletely specified functions using BDDs [Minato, 1992]. Existing methods for SOP generation using SAT solvers are based on the enumeration of satisfying assign-

ments [Morgado and Marques-Silva, 2005]. Sapra et al. [2003] proposed using a SAT solver to implement part of ESPRESSO's procedures for SOP minimisation in order to speed them up. But, as they largely follow the traditional ESPRESSO style of SOP minimisation, they operate only on existing SOPs and do not consider generating a new SOP from a multi-level representation of a Boolean function. Moreover, its runtime and end results significantly depend on the SOP received as input.

A novel SAT-based algorithm for generating irredundant SOPs

Accordingly, our main contribution in this chapter is our second SAT-based method—a new engine for SOP generation and minimisation, completely based on SAT solvers. Our method generates an SOP progressively, building it cube by cube. We guarantee that the generated SOPs are *irredundant*, meaning that no literal and no cube can be deleted without changing the function. As we show with the experimental results, our algorithm generates SOPs with the size similar to that of the BDD-based method [Minato, 1992]. Interestingly, for some circuits, we generate smaller SOPs (up to 10%), which is useful in practical applications. For example, when a multi-level description of the circuit is built using an SOP produced by the proposed SAT-based method, the area-delay product of the resulting circuit, assuming unit-area and unit-delay model, often decreases (up to 27%), compared to global restructuring using BDDs.

Two main features characterise our SAT-based SOP generation and make it desirable in various domains.

Benefits of the progressive generation of SOPs

First, we generate an SOP *progressively*, unlike BDD-based methods that attempt to construct a complete SOP at once. The progressive computation enables generating *a partial SOP* for circuits whose complete SOP cannot be computed given the resource limits. The partial SOPs can be exploited by other applications that do not require the complete circuit functionality, but work with partially defined functions [Chang et al., 2010; Verma et al., 2009]. Moreover, for circuits with large SOPs, the progressive generation enables predicting whether it is feasible to build an SOP for a circuit, and checking if the SOP size is within the limits of the methods that are going to use it. For this, at any moment, we can retrieve the number of outputs for which the SOP is already computed, as well as the finished SOP portion of the currently processed output. We can also easily compute an estimate or a lower-bound of the percentage of covered minterms, considering a uniform distribution of minterms

in the space or considering the size of the truth table, respectively. In contrast, the termination time and the quality of results of the BDD-based methods are unpredictable because the complete BDD has to be built before converting it to an SOP.

Generation of canonical SOPs by using LEXSAT

Second, for the first time, we show that a SAT-based computation can generate *canonical* irredundant SOPs. To this end, we combine (1) the LEXSAT algorithm presented in Chapter 3 that, under a given variable order, generates consecutive satisfying assignments in a lexicographic order, and (2) a deterministic algorithm that expands the received assignments into cubes. For a given function and a variable order, the assignments (i.e., the minterms) are always generated in the same order, and each assignment always results in the same cube. Thus, the resulting SOP is canonical: it is unique and independent of the input implementation of the function. The canonical nature of the resulting SOPs can be useful in the domains where previously only BDDs could be used. For example, as suggested in Section 3.2, canonical SOPs can be used for caching of intermediate results, similarly to how BDDs are used. Also, canonicity brings regularity in the SOPs, hence the results after using algorithms for factoring [Rajski and Vasudevamurthy, 1992] are in some cases better.

The rest of the chapter is organised as follows. First, we describe our algorithm for SAT-based progressive generation of irredundant SOPs in Section 4.1. In Section 4.2, we give our experimental setup and discusses the experimental results. Finally, we conclude and present ideas for future work in Section 4.3. The required background information is provided in Section 2.1 and Section 2.3.

4.1 SAT-Based SOP Generation

Using the algorithm for multi-output circuits, and for incompletely specified functions

In this section, we describe our SAT-based algorithm that progressively generates an irredundant SOP for a single-output function. For multi-output circuits, each output is treated separately. In this chapter, we focus on completely specified functions, but the algorithm can be easily used for incompletely specified functions by providing both the on-set and off-set as an input to the algorithm. In the case of a completely specified function, one of them is derived by complementing the other. The given SAT-based formulation works for incompletely specified func-

tions without any changes. Indeed, after extracting the first cube and blocking it in the on-set of the function, the rest of the computation is performed for an incompletely specified function, even if the initial function was completely specified.

Overview of the SAT-based SOP generation

The presented algorithm iteratively generates minterms, expands them into prime cubes, and adds these cubes to the SOP. The SAT-based heuristics for minterm generation and cube expansion are described in Section 4.1.1 and Section 4.1.2, respectively. Finally, to guarantee that the resulting SOP is irredundant, it is post-processed to remove redundant cubes, as described in Section 4.1.3. Additionally, Section 4.1.4 describes several techniques that reduce the runtime.

Different options for the number of used SAT solvers

The algorithm can be implemented with one SAT solver parameterised to store both on-set and off-set. Alternatively, it can use two solvers, one for on-set and one for off-set, which for convenience we call *on-set SAT solver* and *off-set SAT solver*, respectively. In our implementation of the algorithm, we use four different SAT solvers: for both on-set and off-set, one is used to generate satisfying assignments, the other to expand assignments to cubes. By employing four solvers, we ensure that assignment generation and expansion do not interact with each other during the SOP computation.

Computation of on-set and off-set SOPs

The procedures described in the following subsections assume that we are generating the on-set SOP. The same procedures are used to generate the off-set SOP, by substituting the on-set SAT solver with an off-set SAT solver and vice versa.

4.1.1 Generation of Minterms

Checking if the function is a constant

In order to generate minterms for the on-set of a function f, we initialise a SAT solver with the CNF of f. Then, to discard the trivial case when the function has a constant on-set, we solve the SAT problem by asserting that $f = 0$. If the problem is UNSAT, then f is a constant 1, and we return an SOP with one constant cube. Otherwise, if the problem is SAT, we continue with one of the following methods for minterm generation. Figure 4.1 shows the flowchart of these methods, as well as their connection with the other methods of the SOP generation algorithm.

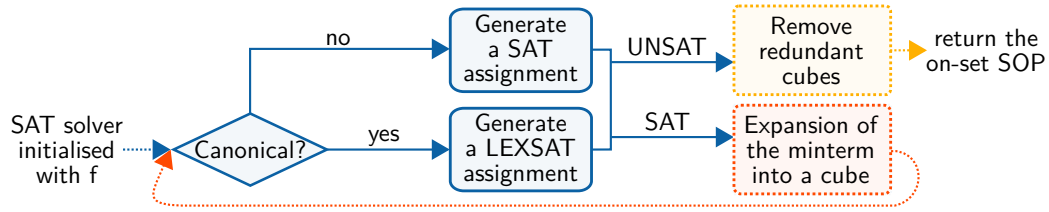


Figure 4.1 – Flowchart of the algorithm for minterm generation. Minterms are generated either as satisfying or LEXSAT assignments. If the problem is satisfiable, the generated minterm is passed to the cube expansion algorithm to generate a cube that would cover the minterm. Once all minterms are covered by the generated cubes, the SAT problem becomes UNSAT, and the SOP is returned after removing the redundant cubes.

Generation of non-canonical SOP

To generate an on-set minterm, we call the SAT-solving procedure of an on-set SAT solver. If the problem is SAT, an assignment for the inputs is returned for which the function evaluates to 1. From the assignment, we can generate a minterm for the function f in which the variables assigned to 0 and 1 are represented with the negative and positive literal, respectively. For example, for a function $f(x, y, z)$, the assignment 110 implies the minterm $xy\bar{z}$. Once a minterm is obtained, we expand it into a cube using the heuristic procedure from Section 4.1.2. Next, we add the cube with its literals complemented to the SAT solver as a *blocking clause*, which is an additional clause that blocks known solutions of the SAT problem. Thus, the next call of the SAT-solving procedure generates a new minterm that is not covered by any of the previously generated cubes. As long as the problem is SAT, we iteratively obtain a minterm, expand it to a cube, and add the cube both to the SOP and to the SAT solver as a blocking clause. The unsatisfiability of the problem indicates that the generated SOP is complete and covers all on-set minterms.

Generation of canonical SOP

Generating minterms from satisfying assignments received from a SAT solver does not guarantee canonicity as SAT solvers return minterms in a non-deterministic order that depends on the design of the solver and the CNF generated for the function. Thus, to obtain canonicity, we iteratively use the binary search-based LEXSAT algorithm `BINARY` presented in Section 3.4.2. The algorithm `BINARY` receives as input a potential assignment that is the lexicographically smallest assignment that might be satisfying. This potential assignment is an assignment with all 0s when generating the first minterm, and afterwards it is assigned to the last generated minterm. Then, `BINARY` tries to verify and fix the assignment of each variable defined with the potential assignment start-

ing from the leftmost variables and moving to right. We also use the proposed methods for runtime improvement, which are presented in Section 3.4.3, fixing the leading 1s, correcting the initial potential assignment, and profiling the success of the first SAT call. Similarly to the non-canonical SOPs, once we obtain a minterm, we expand it into a cube and add it to the SAT solver as a blocking clause.

Example 4.1.1. For example, assume that for the function $f(x_1,\ldots,x_8)$, the last generated minterm 11000001 is received as an initial potential assignment. As this minterm is covered by the last cube, this assignment is not satisfying, so we can increase its value for 1 to get the smallest assignment that might be satisfying 11000010. Next, we fix the assignments of the leading 1s $x_1 = 1$ and $x_2 = 1$, because the next lexicographically smallest assignment has to start with the same leading 1s. Hence, we should only check the assignments for x_i, for $3 \leq i \leq 8$. Due to using binary search, with the first SAT call we assume half of the unfixed assignments, and we give to the on-set SAT solver the assumptions $(x_1,\ldots,x_5) = (1,1,0,0,0)$. Assume that the problem was satisfiable and the SAT solver returned the assignment 11000011 for the input variables. This assignment proves that an on-set minterm with the assumed values exists, but we can also learn that the assignments from the potential minterm $x_6 = 0$ and $x_7 = 1$ are correct. Next, to check if the assignment for the last input x_8 can be set to 0, we call the SAT solver with the assumptions $(x_1,\ldots,x_8) = (1,1,0,0,0,0,1,0)$. If it returns SAT, we return the potential assignment as a minterm because all assignments are verified and fixed. Otherwise, we flip x_8 to 1 to increase the potential assignment before returning it.

4.1.2 Expansion of Minterms into Cubes

In this subsection, we describe our SAT-based procedure that receives a minterm and transforms it into a prime cube by iteratively removing literals (i.e., substituting them with don't-cares). For the on-set SOP, a literal can be removed, if after its removal all minterms covered by the cube do not overlap with the off-set. Figure 4.2 shows a flowchart of the algorithm.

Canonical expansion to prime cubes

The following deterministic algorithm expands a minterm into a cube by ensuring that, after removing each literal, the cube covers only on-set minterms. As the literals are removed always in the same order, which

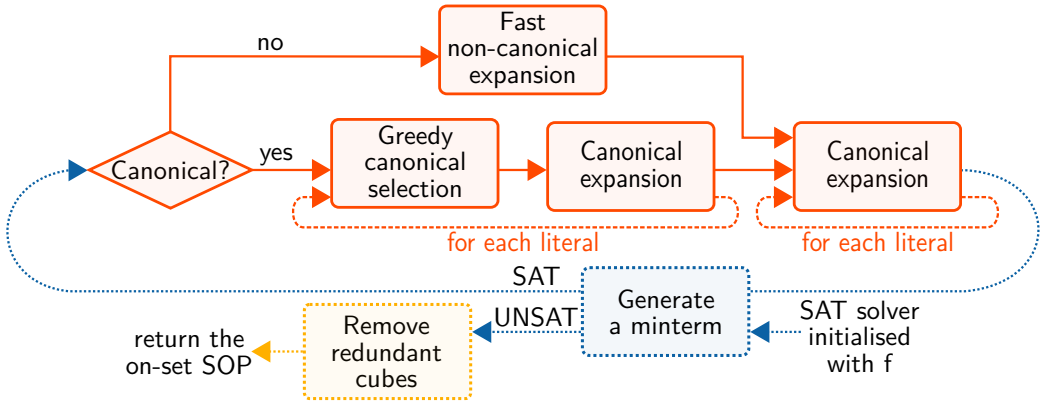


Figure 4.2 – Flowchart of the algorithm for expansion of minterms into cubes. The algorithm for canonical expansion ensures that all generated cubes are prime. After a cube is generated, it is added as a blocking clause to the SAT solver used for minterm generation, and another minterm is generated.

can be specified by the user, the algorithm is deterministic and produces canonical cubes when the given minterms are canonical. Hence, to remove a literal, first, we assume that the literal is removed from the cube, and an off-set SAT solver is run with assumptions for the remaining literals of the cube. On the one hand, if the problem is UNSAT, then no minterm covered by the cube belongs to the off-set, so we can extend the cube by removing this literal. On the other hand, if the problem is SAT, we cannot extend the cube because the SAT solver found an off-set minterm that is covered by the extended cube.

Example 4.1.2. Assume that for the function on Figure 4.3, we received the minterm $\bar{x}y\bar{z}t$. To remove the literal $\bar{x}$, we would call the off-set SAT solver with the assumptions $(y,z,t) = (1,0,1)$. The SAT solver would return SAT, which means that $\bar{x}$ cannot be removed, because the cube $y\bar{z}t$ is covering the off-set minterm $xy\bar{z}t$. However, if we try to remove the literal $\bar{z}$ by calling the SAT solver with the assumptions $(x,y,t) = (0,1,1)$, then we would receive UNSAT because there are no off-set minterms that satisfy these assumptions, so $\bar{z}$ can be removed to obtain the on-set cube c_1.

Greedy canonical literal selection and cube expansion

To minimise the overlapping of cubes, we propose to remove literals in two rounds. In the first round, they are removed greedily, after ensuring that multiple on-set minterms are covered by expanding each literal.

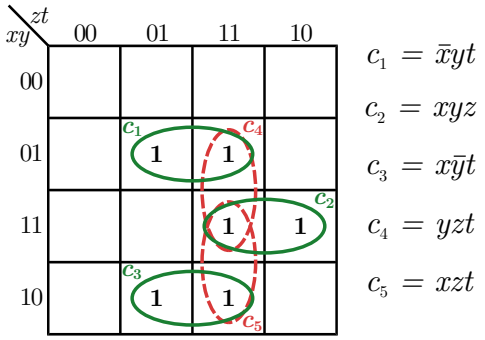


Figure 4.3 – An example for minimum SOP. A Karnaugh map shows the Boolean function $f(x,y,z,t) = \bar{x}yt + xyz + x\bar{y}t$ with its prime cubes c_i, where $1 \leq i \leq 5$. The cubes c_1, c_2 and c_3 are essential and they compose the minimum SOP of f.

Example 4.1.3. Assume that for the function on Figure 4.3, the cube c_1 was computed and added to the on-set SAT solver as a blocking clause. Also, assume that as a second minterm $xyzt$ is generated, which can be extended by removing one of the literals x, y or t. If we remove x, we will obtain the cube c_4 that covers only one additional minterm with respect to the existing cube c_1; but if we remove y or t, we will obtain c_2 or c_5, respectively, each of which covers two yet uncovered minterms.

First round of expansion for generating cubes that cover more than one new minterm

For the minterm $xyzt$ in Example 4.1.3, our expansion procedure skips the opportunity to remove the literal x and, if possible, tries to expand other literals. This greedy selection of literals decides to candidate a literal l_i for removal, if by removing it, the expanded cube covers more than one new minterm. To check if this condition is satisfied, we flip l_i and provide it, along with the remaining literals of the cube, as an assumption to an on-set SAT solver in which the already generated cubes are added as blocking clauses. If the problem is UNSAT, then we skip removing it temporarily. Otherwise, if the problem is SAT, then we consider this literal for removal because by removing it we cover more than one uncovered minterm. Once a literal is a candidate for removal, we run the algorithm for canonical expansion described above to ensure that it can be removed.

Second round of expansion for obtaining prime cubes

However, in this first round, we might skip some opportunities for expansion. Hence, in the second round, for each skipped literal, we execute the algorithm for canonical expansion. This guarantees that, after the second round, no literal can be further removed, which means that

the cube is prime. As we always try to remove literals in the same order, this method generates a canonical SOP.

Fast non-canonical expansion

If generating a canonical SOP is not required, we can substitute the first round of expansion with a faster method to improve runtime: If in an off-set SAT solver we assume the values from the received on-set minterm, the problem is UNSAT and the SAT solver returns the set of assumptions for UNSAT. As the returned literals are sufficient to prove unsatisfiability in an off-set SAT solver, they construct a cube that covers only on-set minterms, and we can remove literals that are not returned by the SAT solver. However, the set of remaining literals is not always minimum, hence we run additionally the algorithm for canonical expansion as a second round to obtain a prime cube.

4.1.3 Removing Redundant Cubes

The initial SOP might contain redundant cubes

The cubes expanded with the methods from Section 4.1.2 are prime by construction. However, by progressively adding cubes to the SAT solver, as described in Section 4.1.1, we ensure that each cube is irredundant with respect to the preceding cubes, but not with respect to the whole set of cubes.

Example 4.1.4. For the function f from Figure 4.3, assume that the cubes c_1, c_5, c_2 and c_3 are generated in the given order. The cube c_5 is irredundant with respect to c_1, because it additionally covers the minterms $xyzt$ and $x\bar{y}zt$, but it is contained in the union of c_2 and c_3.

An algorithm for removing redundant cubes

In order to produce an irredundant SOP, after generating all cubes, we iterate through the cubes to detect and remove redundant ones. First, we initialise a new SAT solver with clauses for all generated cubes and we assume that all cubes are required. Then, by using the assumption mechanism, for each cube c_i, we check if there is an assignment for which c_i evaluates to 1 while all the other irredundant cubes evaluate to 0. If the problem is SAT, the cube is irredundant and the SAT solver returns an assignment that corresponds to a minterm that is covered only by c_i. Otherwise, if the problem is UNSAT, then the cube is redundant, and it is thus removed from the SOP and is excluded when checking the redundancy of the following cubes. As we always try to remove cubes in the order in which they were generated, this method is deterministic and maintains canonicity when canonical SOPs are generated.

Example 4.1.5. Considering the cubes from Example 4.1.4, to check whether c_3 is redundant, we set $c_3 = 1$ by assuming the values $x = 1$, $y = 0$ and $t = 1$. For the assumed values, the other cubes evaluate to $c_1 = 0$, $c_2 = 0$ and $c_5 = z$. Setting $z = 0$ leads to $c_5 = 0$. Thus, the problem is SAT and c_3 is irredundant. The returned assignment 1001 defines the minterm $x\bar{y}\bar{z}t$ that is covered only by c_3.

4.1.4 Improving the Runtime

In this subsection, we present four techniques that improve the runtime of the algorithm by enabling early termination and by treating some special cases.

Simultaneous generation of an on-set and an off-set SOP

For a given function, the SOPs of the on-set and off-set often differ in size, where the SOP size is equal to the number of cubes in it. For example, a three-input function implementing an AND gate, $f(x, y, z) = xyz$, has an on-set SOP, $f = S_{\texttt{on}} = xyz$ with size 1, and an off-set SOP, $\bar{f} = S_{\texttt{off}} = \bar{x} + \bar{y} + \bar{z}$ with size 3. As we want to use the set with a smaller SOP, we simultaneously generate two SOPs, for both the on-set and off-set, by generating one cube at a time from each set. The generation of cubes stops as soon as one SOP is complete. This way, if one set is much smaller than the other, we can avoid the situation when the larger set of cubes has to be completely generated before the smaller set is discovered.

Prioritising outputs with large SOPs

For multi-output circuits, before generating an SOP for each output, we propose to sort outputs by size of their input supports. The outputs with larger supports are processed first, as it is more likely that the SOP generation for these outputs will exceed resource limits, so we can determine if we should terminate the computation earlier.

Benefit from the structure sharing: dividing the outputs into isomorphic classes

To benefit from the structure sharing among the circuit outputs, we implemented a method that decreases the runtime by detecting isomorphic outputs. For this, first, we divide the outputs into isomorphic classes. Two outputs are *isomorphic* and belong to the same class, if they implement an identical function using different inputs. Then, for each class, we generate an SOP only for one output, the class representative, and we duplicate it for the others. In Section 4.2.2, we show that this leads to effective generation of an SOP only for 16.5% of all combinational outputs and has a big influence on scalability.

Benefit from the logic sharing: generation of a single CNF for all outputs

Generating a CNF for each output is time consuming. Therefore, to benefit from the logic sharing among the outputs, we can optionally share one CNF that corresponds to the complete circuit. For this, first, we generate the CNF of the circuit; then, for each output, we initialise a SAT solver only with the part of the CNF for the corresponding output. Besides improving the runtime, as Table 4.1 shows, this option sometimes leads to better results in terms of area-delay product after global restructuring.

Opportunities for runtime improvement by exploiting parallelism

There are several opportunities where computations are independent and can be parallelised. First, the computation of the on-set and off-set SOPs can be executed in parallel. Because now we compute sequentially one cube for each SOP interchangeably, it is expected that this would decrease the runtime by 2x. Second, instead of computing the SOP of each output one after the other, we can also compute each of them in parallel. Finally, for one SOP, we can compute cubes in parallel by generating minterms from different parts of the Boolean space. However, in this chapter, all computations are done sequentially. Analysing and exploiting the effect of parallelism is left for future work.

4.2 Experimental Results

In this section, we describe our experimental setup and compare the proposed SAT-based algorithm with the state-of-the-art BDD-based method.

4.2.1 Experimental Setup

Command from ABC used for the experiments

We implemented the SAT-based algorithm described in Section 4.1 as a new command *satclp* in ABC [ABC]. ABC features an integrated SAT solver based on an early version of MiniSAT [Eén and Sörensson, 2003] that supports incremental SAT solving. Furthermore, ABC provides an implementation of the BDD-based method for SOP generation, specifically the BDD construction for a multi-level circuit (command *collapse*) and the BDD-based ISOP computation [Minato, 1992] (command *sop*). For convenience, in this section, we refer to the SAT-based and BDD-based methods as `SATCLP` and `BDDCLP`, respectively. Finally, ABC enables us to analyse the area-delay results when the generated SOPs are used to build a new multi-level circuit structure. A multi-level network

is generated using the *fx* command [Rajski and Vasudevamurthy, 1992]. The network is next converted into an AIG (command *strash*) and optimised with the *dc2* command. The area and delay of the resulting AIGs are compared for different SOP generation methods.

Benchmarks

To evaluate our algorithm, we use the ISCAS'89 benchmarks, a set of large MCNC benchmarks, a set of nine logic tables from the instruction decoder unit [BenchIBM], denoted as LT-DEC, and a set of proprietary industrial benchmarks. The LT-DEC suite is well-suited to demonstrate the factoring gains as circuit size increases [Kravets, 2015]. The names of the LT-DEC benchmarks are given in the form "[$N_{\mathtt{PI}}$]/[$N_{\mathtt{PO}}$]", where $N_{\mathtt{PI}}$ is the number of primary inputs and $N_{\mathtt{PO}}$ is the number of primary outputs. For the main experiments, we discard benchmarks for which the SOP size exceeds the built-in resource limits of the used commands, hence, we use 30 (out of 32) benchmarks from the ISCAS'89 set, 15 (out of 20) benchmarks from the MCNC set, and 17 (out of 18) industrial benchmarks. With the discarded benchmarks, we demonstrate the generation of partial SOPs.

4.2.2 SAT-Based vs. BDD-Based SOP Generation

To analyse the performance and quality of results of the algorithm presented in Section 4.1, we run both `SATCLP` and `BDDCLP` available in ABC. In this section, we present the results of these experiments.

Generating multiple SOPs with each method

Although the command *collapse* dynamically finds a good variable order for the BDD, changing the initial order of the primary inputs results in a different BDD structure, which leads to a different SOP. Hence, to obtain a good SOP, we generate five SOPs for `BDDCLP` by using five different initial orders of the primary inputs. Similarly, `SATCLP` generates different SOPs for different orders of the primary inputs, which define the order of removing literals from the cubes. We either use the pre-defined order from the benchmark file or order the inputs based on their number of fanouts (option "Order PI"), which currently works only for the combinational benchmarks. We can also, optionally, reverse the selected variable order (option "Reverse PI"). Moreover, we can enable generation of canonical SOPs (option "Canonical"); and for non-canonical SOPs we can enable the generation of one CNF for all outputs as described in Section 4.1.4 (option "Shared CNF"). Thus, by changing these four options, we generate 12 SOPs using `SATCLP`.

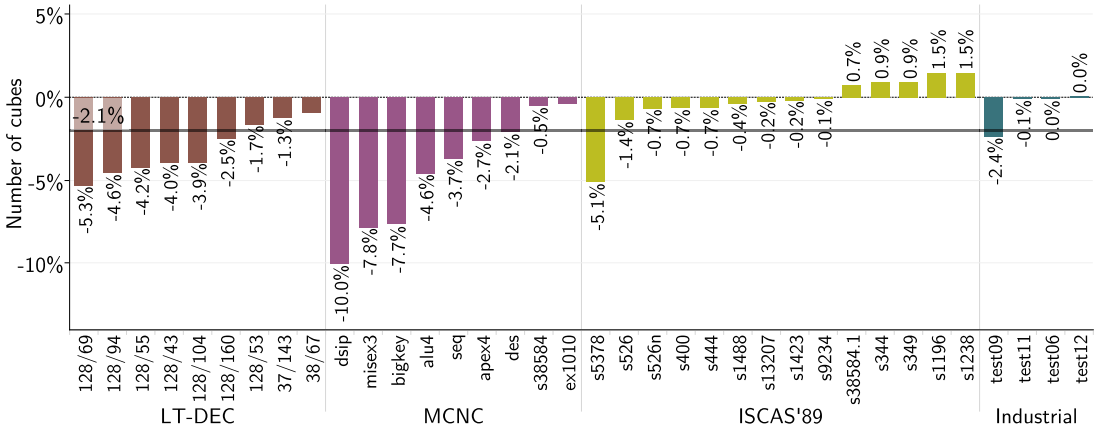


Figure 4.4 – Difference in the size of the smallest SOP generated by each method. The size of the smallest SOPs generated by SATCLP is compared to the size of the smallest SOP generated by BDDCLP. Only the benchmarks for which the SOP size differs are shown. The gray line shows that, on average, SATCLP decreases the SOP size by 2.1%.

Table 4.1 – Number of benchmarks for which activating or deactivating an option for SATCLP results in the smallest SOP in terms of number of cubes or the best area-delay product. In total, 71 benchmarks are used. The number of cases with the smallest SOP in terms of number of cubes is given in the columns under "#Cubes", and the number of cases with best area-delay product is given in the columns under "Area·Delay". If for one benchmark, an identical best result is obtained both when the option is activated and deactivated, then we count it as a tie.

	#Cubes			Area·Delay		
	No	**Yes**	**Tie**	**No**	**Yes**	**Tie**
Canonical	7	34	30	28	26	17
Shared CNF	43	1	27	40	13	18
Order PI	45	8	18	57	11	3
Reverse PI	20	15	36	28	21	22

Comparing the size of the generated SOPs

Generating multiple SOPs with each method results in SOPs that differ in size, where the SOP size is equal to the number of cubes that constitute the SOP. Figure 4.4 shows and compares the benchmarks for which the size of the smallest SOP generated by each method is different. Although SATCLP most often generates SOPs with almost the same size as those generated by BDDCLP, for some benchmarks it generates smaller SOPs (up to 10%). Because the results for SATCLP are obtained using several options, as Table 4.1 shows, under "#Cubes", the number of benchmarks for which the smallest SOP is generated when a given

Table 4.2 – Comparison of the number of combinational outputs in the used benchmarks and the number of isomorphic classes. The combinational outputs are either primary outputs or latch inputs. The number of isomorphic classes is equal to the number of calls of the SAT-based algorithm for SOP generation.

Set	Number of benchmarks	Combinational outputs	Isomorphic classes	Ratio
LT-DEC	9	788	686	87.1%
MCNC	15	3024	1435	47.5%
ISCAS'89	30	5753	1709	29.7%
Industrial	17	64267	8356	13.0%
Total	**71**	**73832**	**12186**	**16.5%**

options is deactivated or activated. We observe that, for 34 benchmarks we obtain an exclusively smaller SOP when generating canonical SOPs, and only for 7 benchmarks smaller are the non-canonical SOPs. Similarly, the SOP size increases for about 60% of the benchmarks when either the CNF is shared or the inputs are ordered by their number of fanouts.

Setup for runtime comparison

Next, we compare the runtimes of the algorithms. The reported runtime is averaged over three runs of the algorithm for SOP generation. For `BDDCLP`, we report the time required to execute the commands *collapse* and *sop*. For `SATCLP`, we report the time taken by our command *satclp*, which includes the time to generate isomorphic outputs, derive CNF, and initialise SAT solver instances, as well as the time for all SAT calls for minterm generation, cube expansion, and removing redundant cubes.

Runtime comparison

In terms of scalability, in Section 4.1.4, we suggested filtering out structurally isomorphic outputs. Due to this, as Table 4.2 shows, an SOP is computed only for 16.5% of the combinational outputs, one for each isomorphic class, and for the other outputs, we duplicate the generated SOP of the class representative. This reduces the runtime of our algorithm `SATCLP`, and for benchmarks rich in isomorphic outputs, the proposed method is significantly faster than `BDDCLP`. For example, from the public benchmarks, the maximum speedup is achieved for the benchmark s35932 from the ISCAS'89 set, for which we generate SOPs only for 10 out of 2048 combinational outputs, hence, on average, `SATCLP` requires 0.10 seconds, whereas `BDDCLP` requires 1.57 seconds. Yet, on average, our `SATCLP` is 7.5x slower than `BDDCLP` for the pub-

Table 4.3 – Runtime results for the combinational industrial benchmarks when SOPs are generated with BDDCLP and SATCLP. The columns "PIs" and "POs" give the number of primary inputs and outputs, respectively. A dash (-) denotes that the method fails to compute an SOP. Highlighted are the cases when SATCLP outperforms BDDCLP.

	PIs	POs	Isomorphic classes	Runtime (s)		
				BDDCLP	SATCLP	
					Non-canonical	Canonical
test01	2513	2377	2083	31.14	165.99	1658.92
test02	3236	3202	3146	-	32.46	112.15
test03	1542	514	113	10.64	12.74	70.79
test04	37397	292	155	144.57	15.01	197.71
test05	1178	606	95	-	141.85	748.81
test06	1488	1446	580	4.24	31.50	137.74
test07	8087	335	270	152.42	17.91	68.31
test08	438	512	432	3.96	17.34	84.67
test09	870	1636	792	2.36	18.17	125.19
test10	2376	1233	314	100.83	10.55	46.88
test11	3875	3274	138	14.49	2.49	7.95
test12	4626	3708	112	10.29	1.59	3.17
test13	1110	1040	74	50.86	1.30	9.29
test14	8514	1323	890	-	-	-
test15	47356	4136	21	-	0.21	0.26
test16	58382	18433	9	-	0.63	0.28
test17	68620	17411	19	-	0.64	0.33
test18	36900	4112	3	603.86	277.08	42292.50
Average (relative to BDDCLP)				**1.00**	**0.54**	**3.76**

lic benchmarks. We have observed that the functions for expanding minterms into cubes are the bottleneck. For example, for the LT-DEC benchmarks, on average, 85% of the runtime is spent in this operation, 8% is spent on minterm generation, 2% on removing redundant cubes, and 5% on other operations, such as dividing the outputs into classes, generating CNF, and initialising SAT solver instances.

Runtime comparison for the industrial benchmarks

Conversely, from Table 4.3, we can see that SATCLP is definitely more scalable than BDDCLP for a suite of proprietary industrial benchmarks. First, it is most often faster than BDDCLP for the benchmarks for which BDDCLP computes an SOP; but it also completes on most test-cases for

which BDDCLP fails. For the non-canonical SOPs, SATCLP decreases the runtime of SOP generation by 45.9%, on average, considering only the benchmarks for which BDDCLP generates an SOP. For canonical SOPs, although SATCLP is 4.3x slower than its non-canonical version and 3.8x slower than BDDCLP, it successfully generates SOPs for 5 benchmarks, for which BDDCLP fails. There is no benchmark for which BDDCLP computes an SOP and SATCLP fails.

SATCLP is more scalable because it computes an SOP for each output separately

The increased scalability of SATCLP is largely due to the fact that most of the industrial testcases have hundreds of inputs and outputs, which makes constructing global BDDs in the same manager problematic for all outputs at once. The algorithm SATCLP does not suffer from this limitation, because it computes the SOPs for one output at a time. It can be argued that the BDD-based computation can also be performed on the per-output basis. However, in this case, the BDD manager will inevitably find different variable orders for different outputs, which will increase the size of the resulting multi-level circuits when these SOPs are factored. In fact, factoring benefits from computing SOP with the same variable order that facilitates creating similar combinations of literals in different cubes, which in turn helps improve the quality of shared divisor extraction and factoring.

Generation of partial SOPs

Finally, as SATCLP generates cubes progressively, unlike BDDCLP, it can build partial SOPs even for large circuits, and these can be used for incremental applications. Figure 4.5 shows the number of cubes that compose the partial non-canonical SOPs when a time limit for the runtime is set to t seconds, where t is an integer value such that $1 \leq t \leq 10$. For functions with larger supports, SATCLP usually generates less cubes because more time is required for cube expansion. Only for the benchmark test14, SATCLP does not generate any cube in the first six seconds due to the large support set of the first processed output that depends on 6246 inputs. For the other benchmarks, SATCLP generates thousands of cubes in just a few seconds. In this experiment, SATCLP still generates at the same time both the on-set and the off-set SOP. However, in the incremental applications, we can generate just one of them, which would increase the number of generated cubes for a given time limit.

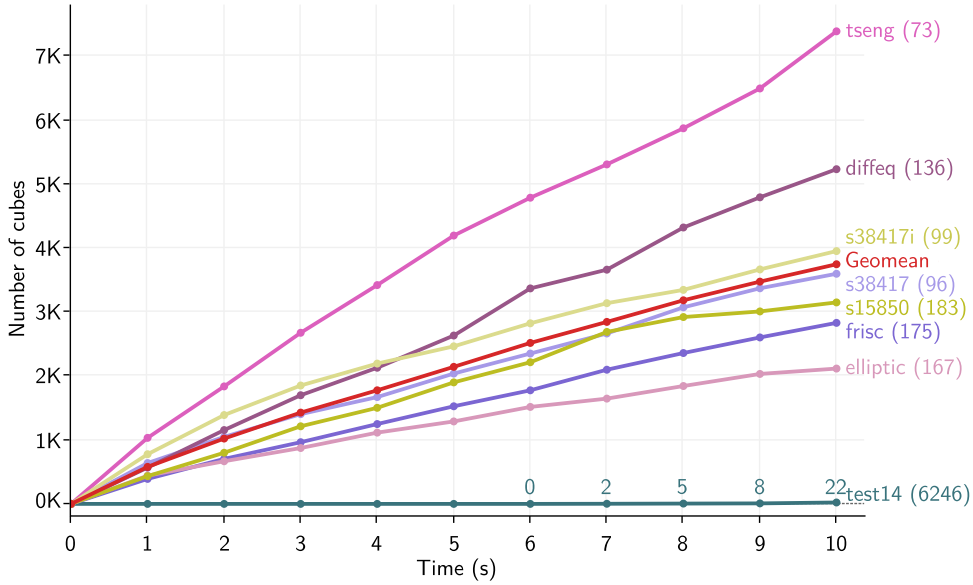


Figure 4.5 – The number of generated cubes in the partial SOPs when the time limit is set between 1 and 10 seconds. The number of generated cubes by SATCLP depends on the size of the support set of the output with the largest support set, which is given in brackets. For all benchmarks, the generated cubes belong to one output.

4.2.3 Case-Study: SAT-Based SOPs for Multi-level Implementation of Circuits

Quality of results when the SOPs are used to generate multi-level circuits

As explained in Section 4.2.2, several SOPs are generated with each method. Different SOPs result in multi-level networks with different areas and delays. As Figure 4.6 shows, for most benchmarks, our algorithm obtains Pareto-optimal solutions, compared to BDDCLP. To obtain these results, we isolate the best circuit implementations in terms of area-delay product as derived by each method. Table 4.1, with the columns "Area·Delay", shows the number of benchmarks with the smallest area-delay product generated when a given option is deactivated and activated for SATCLP. We generate a circuit structure with a smaller area-delay product for 26 benchmarks when generating canonical SOPs; but for 28 benchmarks the non-canonical SOPs are a better option. Also, for most benchmarks it is best to generate an SOP by using the original ordering of primary inputs used in the benchmark file.

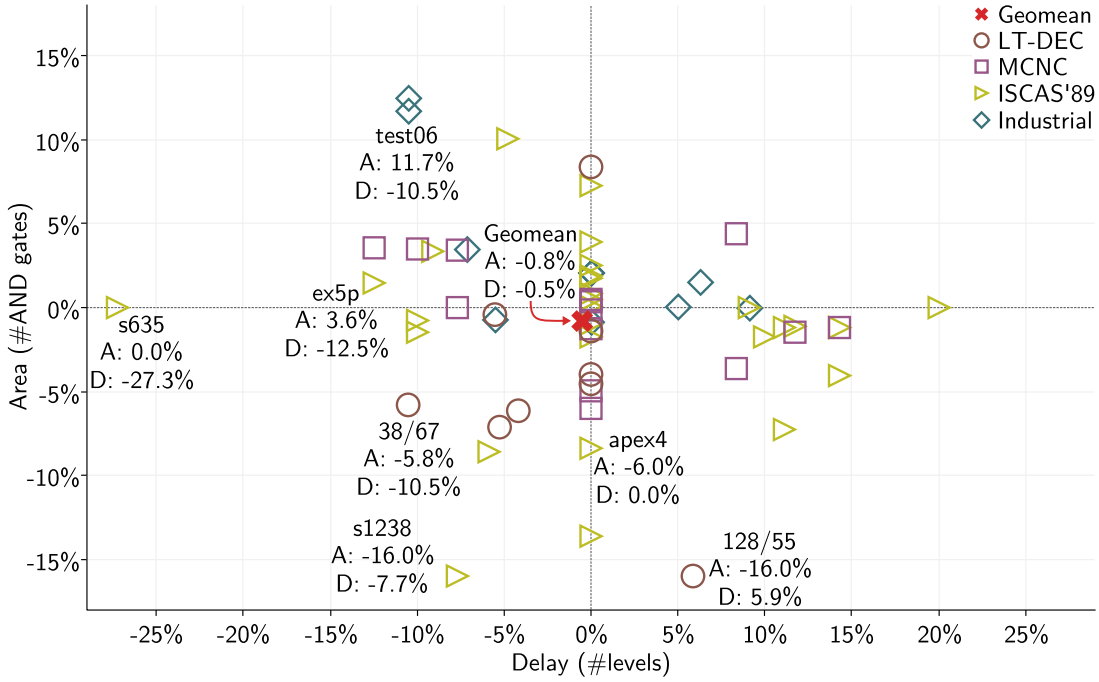


Figure 4.6 – Comparison of the best circuit implementations, in terms of area-delay product, derived by each method. For each benchmark, the best results after a multi-level description is built from the SOPs generated by our SAT-based algorithm are compared to the ones obtained from the BDD-based SOPs. For most benchmarks, we obtain Pareto optimal solutions.

4.3 Conclusion

Key insights

In this chapter, we have presented a novel algorithm for progressive generation of irredundant SOPs using heuristics based solely on SAT solving. The LEXSAT algorithm from Chapter 3 enables the first-time canonical SOPs generated with a SAT solver that are unique for a given function and a variable order. Moreover, the canonicity and the progressive generation make our heuristics desirable for many applications where minterms, cubes, or SOPs are required, and for which the existing methods are either unscalable or impractical to use. Further, in Chapter 6, we have compared a simplified version of the proposed algorithm for SOP generation to algorithms based on a resubstitution miter [Mishchenko et al., 2011b] and interpolation when they are used for resubstitution.

Regarding the quality of results, we have shown that for computing a complete SOP, on average, the SAT-based computation is as good as the BDD-based one. Moreover, the multi-level circuit structures derived using the SOPs generated by our approach are often better or Pareto-optimal.

Regarding the runtime, the proposed method is somewhat slower than the BDD-based method for most of the public benchmarks, but it is faster for circuits that are rich in isomorphic outputs. Thus, for the industrial benchmarks, our method is both faster and more scalable, and therefore it is a good candidate for global circuit restructuring at least in that particular industrial setting.

Ideas for future work

Besides the described opportunities for parallelisation, the proposed method can also benefit from the ongoing improvement in modern SAT solvers. For example, recently we explored a new push/pop interface for assumptions used in the incremental SAT solving, which led to additional runtime improvements. As we show, for some circuits the results can improve by changing the variable order in which the cubes are expanded, but a careful study of this problem is required to improve further the quality of results.

In addition to runtime improvements, future work can be focused on developing a dedicated SAT-based multi-output SOP computation, which computes cubes that are shared between several outputs. A recent publication [Kravets, 2015] indicates that a significant improvement in quality (more than 10%) can be achieved by computing and factoring multi-output SOPs. We are not aware of a practical method for BDD-based multi-output SOP computation, so it is likely that a method based on SAT solving might be the only efficient solution. Another direction for future work is exploring the benefits of the progressive generation of canonical minterms and cubes in different areas. One such area is multi-level logic synthesis where incremental SAT-based decomposition methods can be developed based on partial SOPs computed for the output functions.

5 Constrained Interpolation for Guided Logic Synthesis

Applying Craig interpolation for reimplementation of functions

The success of the SAT-based generation of Craig interpolants in model checking [McMillan, 2003] inspired its use in a variety of logic synthesis applications [Jiang et al., 2010; Lee et al., 2008; Lin et al., 2008; Mishchenko et al., 2011b; Tang et al., 2011]. One of these applications reimplements a target function f as a function of a given set of base functions G [Jiang et al., 2010], which we call *standard interpolation method* for convenience. In this case, the interpolant represents the dependency function h, such that $f = h(G)$. In some situations, the set G contains enough base functions to enable the existence of multiple dependency functions whose quality mainly depends on the base functions selected for the reimplementation. The interpolation is not an optimisation problem, hence, it often omits some base functions that might be required for an optimal implementation of the target function. Mainly, it is impossible to impose that an interpolant uses a specific base function.

A novel interpolation method that imposes the use of a specific base function

In this chapter, we propose our third SAT-based method—a new *carving interpolation method*—that overcomes this specific limitation of the standard interpolation method. The carving interpolation can impose a specific base function $g_i \in G$ as a primary input of the generated dependency function. Such a dependency function is built as a Shannon expansion of two constrained Craig interpolants for the assignments of the primary inputs for which g_i evaluates to 0 and 1, respectively. We also introduce a method that iteratively imposes, one by one, a given set of base functions. In each iteration, we impose a base function by

This chapter is based on the work of a paper published at the 2014 International Conference on Computer Aided Design [Petkovska et al., 2014].

generating a new dependency function that is used as a target function for the next iteration. This new feature can be particularly useful for some synthesis-based algorithms that reconstruct circuits.

Potential benefits for ECO algorithms

Some synthesis-based *engineering change order (ECO)* algorithms [Tang et al., 2011; Wu et al., 2010] use Craig interpolation to derive logic circuits, called *patches*, that correct a flawed portion of a circuit. To maximise the reuse of logic from the flawed implementation, the patch is built as an interpolant that uses a set of implemented components as the base functions. In order to build the interpolant, the standard interpolation method relies on the structure of the proof of unsatisfiability, which in turn is strongly biased by the heuristics used by the SAT solver. These heuristics are agnostic to the purposes of the ECO engine, hence they offer no control over the selection of components that are used to build the patch and generally do not return the best circuit structure for it. In contrast, our carving method can overcome the deficiencies of these heuristics and result in more compact patches, as it gives better control of the functions to be included in the logic cone of a patch.

Potential benefits for algorithms for global circuit restructuring

Attempts to synthesise logic circuits by performing global restructuring could also profit from our carving technique. Some logic optimisation heuristics [Verma et al., 2009, 2007] reconstruct the input circuit structure by making use of small single-output circuits, called *bricks*, that are found to be useful by a particular utility function. These algorithms construct the output circuit structure gradually using sets of bricks. The standard interpolation method is incompatible with these heuristics, because they require the target function to be recomposed either with some specific bricks from a set or with the complete set. Conversely, the proposed carving technique removes this limitation. Additionally, the original heuristic produce a functionally correct circuit only at the end of the algorithm, whereas our carving technique produces one every time a base function is imposed into the circuit. This feature, in addition to offering a palette of implementations for the input circuit, can assist in tuning the bricks' utility function.

Overview of the results

In this chapter, we compare our carving technique with the standard Craig interpolation method. Our results show that the carving technique is able to successfully include a desired base function 99% of the time, whereas the standard interpolation method has a failure rate up to 59%. We also study the runtime of our techniques.

The rest of the chapter is organised as follows. First, in Section 5.1, we motivate our work with an example. Next, in Section 5.2, we describe the standard interpolation method. In Section 5.3, we describe in detail the proposed carving methods. Finally, we present our experimental results in Section 5.4 and conclude in Section 5.5. The required background information is provided in Section 2.1, Section 2.3, Section 2.4, Section 2.5, and Section 2.6.

5.1 Motivating Example

Reimplementation of the carry function of a 2-bit adder by using Craig interpolation

Assume we have already implemented the sum function of a 2-bit adder

$$s_1 = a_1 \oplus b_1 \oplus (a_0 \cdot b_0)$$

using the base functions $g_1 = a_1 \oplus b_1$ and $g_2 = a_0 \cdot b_0$ as

$$s_1 = g_1 \oplus g_2.$$

Next, we want to reimplement the carry function of the same 2-bit adder

$$c_1 = (a_1 \cdot b_1) + (a_0 \cdot b_0 \cdot (a_1 + b_1))$$

and, for additional base functions, we have only $g_3 = a_1$ and $g_4 = b_1$. We can now use a SAT solver and the standard Craig interpolation method to rewrite the target function c_1 by using the given set of base functions $G = \{g_1, g_2, g_3, g_4\}$. Unfortunately, the standard interpolation method might arbitrarily return any of the three dependency functions

$$h_1 = (g_3 \cdot g_4) + (g_2 \cdot (g_3 + g_4)),$$
$$h_2 = (g_3 \cdot g_4) + (g_2 \cdot (g_3 \oplus g_4)), \text{ or}$$
$$h_3 = (g_3 \cdot g_4) + (g_2 \cdot g_1).$$

However, having already implemented s_1 as above, we might be interested in h_3 that is the area-optimal solution and avoids reimplementing existing logic. In contrast, our carving method obtains exactly h_3 by imposing g_1 as a primary input.

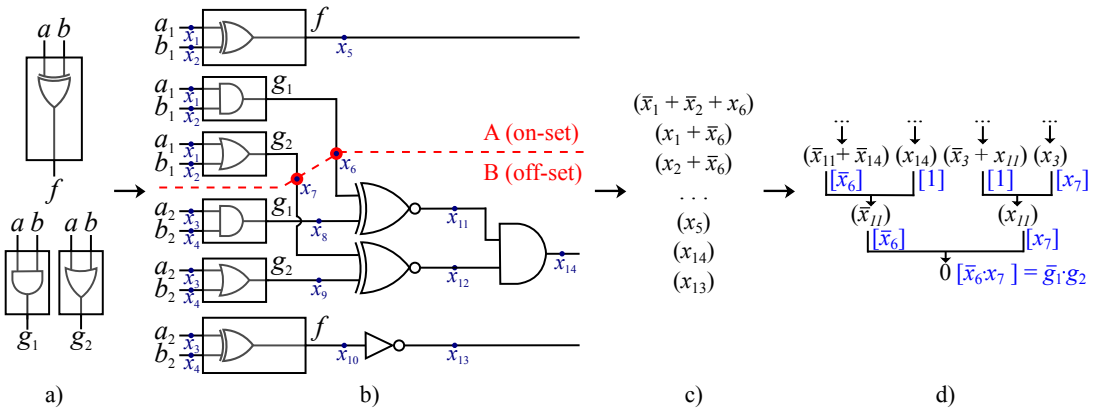


Figure 5.1 – The process for computing a single-output dependency function using the standard interpolation method. The target function $f(a,b)$ and the set of base functions $G = \{g_1, g_2\}$ are shown in Figure 5.1a. Figure 5.1b presents the DLN used for deriving the interpolant. The variable vector $X = (x_1, \ldots, x_{14})$ corresponds to the introduced CNF variables for each signal. The dashed line partitions the gates whose clauses belong to A and B, respectively. The dots on this line depict the outputs of the base functions whose CNF variables are common between the on-set A and the off-set B. Thus, these outputs are candidates for the support set of the dependency function. Figure 5.1c shows the same DLN represented as CNF clauses, which are given to a SAT solver. Figure 5.1d shows the final clauses of the refutation proof from which the dependency function $h(g_1, g_2) = \bar{g}_1 \cdot g_2$ is derived.

Situations qualitatively similar to this one can arise in a variety of logic synthesis situations such as techniques for restructuring complex circuits into well-studied architectures. This example could also represent an ECO problem, if the function s_1 is implemented flawlessly and the function c_1 has to be rectified.

5.2 The Standard Interpolation Method

Generation of a dependency function using the standard interpolation method

Jiang et al. [2010] suggested using interpolation to re-express a target function as some dependency function over other base functions (also used by Lin et al. [2009]). As previously defined, in this chapter, we call this method the *standard interpolation method*. Figure 5.1b shows the *dependency logic network (DLN)* required for computing the dependency function h of the target function f in terms of the set of base functions $G = \{g_1, g_2\}$. The DLN has similar properties as the miter shown in Figure 2.3, and can be used for the functional dependency check of a single-output function. Thus, if f functionally depends on the set of base functions G, then we can represent the DLN as a set of

CNF clauses partitioned into two sets, A and B, as suggested by Jiang et al. [2010]. The Tseitin transformation [Tseitin, 1983] converts a circuit from combinational logic to a set of CNF clauses by introducing new variables for each primary input and for each gate. A SAT solver is initialised with these CNF clauses and it constrains the CNF variables assigned to the two copies of the primary inputs of the target function. As f functionally depends on G, the problem is UNSAT, and the SAT solver returns a proof of unsatisfiability, which is also called refutation proof as defined in Section 2.3. This refutation proof is used to compute the interpolant. As the CNF variables of the outputs of one copy of the base functions belong both to the on-set A and to the off-set B, these outputs represent candidate inputs for the support set of the interpolant and, implicatively, for the dependency function. Figure 5.1 shows the process for computing the dependency function with this standard interpolation method.

5.3 The Carving Interpolation Method

In this section, we first explain the deficiency of the standard interpolation method. Next, we present in detail the proposed carving interpolation method for both a single base function and a set of base functions.

5.3.1 Deficiency of the Standard Interpolation Method

Reason for omitting some base functions

A base function belongs to the support set of the dependency function if and only if its clauses are part of the refutation proof. All essential base functions belong to the support set, because their clauses are required for building a refutation proof. However, whether an auxiliary base function belongs to the support set depends on the selection of the SAT solver. As the SAT solver neglects the importance of a base function when building the refutation proof, the standard interpolation method often omits base functions that enable a specific efficient implementation of the target function.

Illustrating the problem with an application

In contrast, some techniques require either some specific or all selected base functions to be used as primary inputs of the dependency function. For instance, Verma et al. [2009] propose an algorithm that optimises a circuit by iteratively generating and selecting a set of base functions.

After the first iteration, it uses the dependency function to generate a new set of base functions. Thus, if a base function is disconnected from the dependency function, it is then discarded, although it was previously selected as adequate and desirable for reconstructing the circuit. Therefore, this behaviour of the standard interpolation method hinders its applicability to such logic synthesis techniques.

5.3.2 Carving Out a Base Function

In this section, we describe in detail the *carving interpolation method* for imposing the use of a single base function: it constructs a dependency function as a Shannon expansion of two constrained Craig interpolants.

Cofactoring the set of CNF clauses

Assume we have a target function that functionally depends on a set of base functions from which we want to impose the use of a selected base function. For the carving interpolation method, we construct the same DLN as for the standard interpolation method and represent it as a set of CNF clauses. Assume the CNF clauses are expressed in terms of the variable vector $X = (x_1, \ldots, x_n)$. We denote this set of CNF clauses as $C(x_1, \ldots, x_n)$. The unsatisfiability of the SAT problem defined with the DLN signifies that the set of CNF clauses is UNSAT for any assignment of X. From this, it follows that both $C_{\bar{x}_i} = C(x_1, \ldots, 0, \ldots, x_n)$ and $C_{x_i} = C(x_1, \ldots, 1, \ldots, x_n)$, in which we have assigned the variable x_i to 0 and 1, respectively, are also UNSAT for any assignments of X. A CNF variable is assigned to a constant value by extending the existing set of CNF clauses with a single-variable clause x_i or $\bar{x}_i$, which makes an assumption that the variable x_i evaluates to 1 or 0, respectively.

Constructing an interpolant as a Shannon expansion of two constrained interpolants

Using the refutation proofs for $C_{\bar{x}_i}$ and C_{x_i}, we can construct two constrained interpolants $I_{\bar{x}_i}$ and I_{x_i}, respectively. These two interpolants represent the cofactors of a feasible interpolant I for the satisfiability problem expressed with the set of clauses $C(x_1, \ldots, x_n)$, with respect to x_i. Thus, using the formula for Shannon expansion, we can generate the interpolant I as

$$I = \bar{x}_i \cdot I_{\bar{x}_i} + x_i \cdot I_{x_i}.$$

Constructing an interpolant that imposes the use of a single base function

In order to impose a selected base function g_i, we perform the Shannon expansion with respect to the CNF variable x_i assigned to the output of the function g_i. When we assign the variable x_i to 0, we evaluate

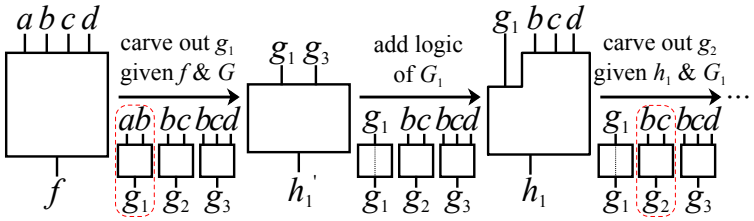


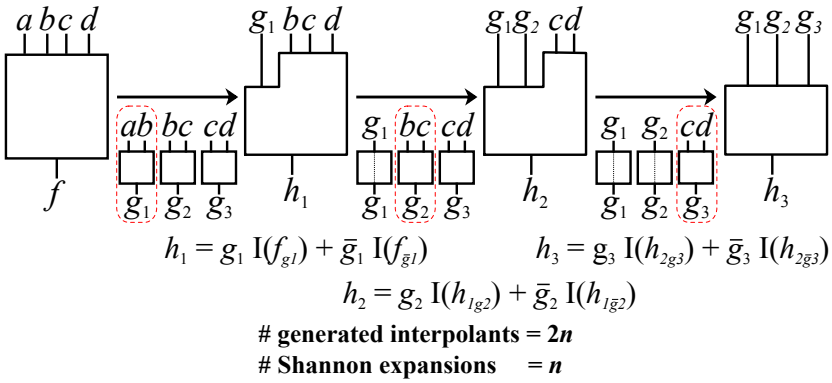
Figure 5.2 – The process for imposing the first base function g_1 from a set of base functions $G = \{g_1, g_2, g_3\}$. After a base function is imposed, it is substituted with an identity function in the set G. To impose the second function g_2, the carving method is given the last computed interpolant h_1 and the modified set of base functions G_1.

all assignments for which the function g_i evaluates to 0. At the same time, for the assignments for which g_i evaluates to 1, there is a conflict with the assumed value of the output of g_i; and the problem is UNSAT. Similarly, the dual case applies for the assignments for which g_i evaluates to 1. Thus, the two interpolants, built for the assignments for which the selected base function g_i evaluates to 0 and 1, respectively, represent the negative and the positive cofactors of the dependency function with respect to x_i, and they can be used to obtain the final dependency function with Shannon expansion.

5.3.3 Carving Out a Set of Base Functions

Iteratively carving out a set of base functions one by one

Given a set of base functions $G = \{g_1, \ldots, g_n\}$, such that the target function f functionally depends on G, the base functions are iteratively carved out one by one. To carve out the first base function g_1, we use the function f and the set G. As a result, we receive a dependency function h_1' that has the imposed base function g_1 and a subset of the remaining base functions from G as primary inputs. To retain g_1 as a primary input, after carving it out, we modify G to G_1 by substituting g_1 with an *identity function* that propagates the input as an output. To be able to impose the remaining base functions, we construct a function h_1 by adding the logic of the non-imposed base functions to the function h_1'. In the next iterations, to impose the base function g_i, where $i = 2, \ldots, n$, we use the dependency function h_{i-1} and the modified set of base functions G_{i-1}. Figure 5.2 shows the first iteration of our method when imposing a set of base functions $G = \{g_1, g_2, g_3\}$ given a target function f.

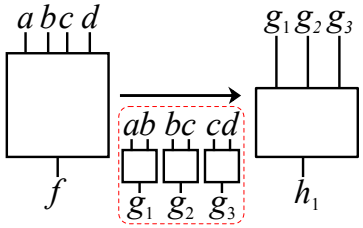


$$h_1 = g_1\,\mathrm{I}(f_{g1}) + \bar{g}_1\,\mathrm{I}(f_{\bar{g}1}) \qquad h_3 = g_3\,\mathrm{I}(h_{2g3}) + \bar{g}_3\,\mathrm{I}(h_{2\bar{g}3})$$

$$h_2 = g_2\,\mathrm{I}(h_{1g2}) + \bar{g}_2\,\mathrm{I}(h_{1\bar{g}2})$$

 # generated interpolants = 2*n*
 # Shannon expansions = *n*

(a) Carving the base functions one by one.



$$h_1 = g_1 g_2 g_3 \mathrm{I}(f_{g1g2g3}) + \ldots + \bar{g}_1\bar{g}_2\bar{g}_3\mathrm{I}(f_{\bar{g}1\bar{g}2\bar{g}3})$$

 # generated interpolants = 2^n
 # Shannon expansions = 1

(b) Carving multiple base functions simultaneously.

Figure 5.3 – Difference between carving multiple base functions one by one and simultaneously. To carve out n base functions successively, we need to compute only $2n$ interpolants. Whereas, to carve them out simultaneously, we must compute 2^n interpolants.

Idea for improving the runtime by carving out only auxiliary base functions

Carving a set of base functions by our method is usually much slower than by the standard interpolation method, because our method builds two interpolants per base function, whereas the standard method generates only one interpolant per set of base functions. Accordingly, as we know that the standard interpolation method always uses the essential base functions, we propose an optimised carving method that improves the runtime by generating a single interpolant for all essential base functions. Furthermore, it increases the success rate for imposing the set of base functions, because it prioritises the auxiliary base functions and carves them out when all or most of the circuit's logic is available.

Optimised carving method for a set of base functions

The optimised carving method starts with partitioning the set G into two subsets, G_e and G_a, that contain the essential and auxiliary base

functions, respectively. Then, we first carve out the base functions from the set G_a, which might be omitted by the standard interpolation method, as explained in Section 5.3.1. Assuming that the set G_a contains i base functions, once all functions are carved out, we constructed the function h_i and modified the set G to G_i. The set G_i contains (1) identity base functions for the functions from G_a and (2) all functions from G_e. Finally, the standard interpolation method is used to construct the resulting dependency function by re-expressing the function h_i as a function from the set G_i.

Example 5.3.1. For the target function f and the set G from Figure 5.2, we first impose the function $g_2 \in G_a$ and obtain the function h_1 instead of first imposing the function g_1 as shown for the basic carving method. Next, instead of imposing the functions from $G_e = \{g_1, g_3\}$, we generate the final dependency function with the standard interpolation method given the function h_1 as a target function and the set $G_1 = \{g_1, g_{2id}, g_3\}$, where g_{2id} is the identity function introduced for g_2.

Carving out multiple base functions simultaneously

Shannon expansion can also be performed on multiple variables. Thus, in one iteration, multiple base functions can be simultaneously carved out of the target function. However, as Figure 5.3 shows, due to the nature of the Shannon expansion, the number of interpolants grows exponentially with the number of base functions carved out simultaneously, in contrast with the linear growth of the one-by-one single-function carving.

5.4 Experimental Results

In this section, we introduce our experimental setup and we present the experimental results that compare our carving methods with the standard interpolation method.

5.4.1 Experimental Setup

Integration of the algorithms in ABC

We implemented both the basic and the optimised carving interpolation methods as commands in ABC [ABC]. An advantage of using ABC is that the integrated SAT solver, which is based on an early version of MiniSAT [Eén and Sörensson, 2003], provides a proof of unsatisfiability for UNSAT problems. ABC also provides an implementation of an algo-

rithm that generates a Craig interpolant from a proof of unsatisfiability as an AIG.

Inspiration from Iterative Layering for the experimental setup

Although the carving technique is general and not tied to any purpose or logic synthesis heuristic, our experimental setup is somewhat inspired from the Iterative Layering technique proposed by Verma et al. [2009]. This heuristic restructures a circuit by gradually imposing a set of preselected base functions. Reimplementing the whole algorithm is not our purpose here, but we want to explore interpolation as the means to progressively transform the original circuit into an optimised one (note that the original implementation of Iterative Layering does not use SAT solvers and follows a different strategy). Hence, in our experiments, we use an oracle that gradually provides the base functions for composing the optimal circuit structure.

Experimental setup

To ABC, we provide an input implementation and a final implementation of the circuit we want to optimise. The final implementation is either a desirable known implementation of the circuit or the input implementation optimised in ABC. It serves as a reference goal from which an oracle computes the set of base functions for reconstructing the circuit. In our case, this set consists of the logic functions of non-overlapping k-input cuts which cover the whole circuit. With this setup, we show that we can recompose any input implementation to any final implementation, as long as we have the adequate base functions. Two scenarios are presented: In the first, we carve out only a single base function, as described in Section 5.3.2; in the second, we carve out a complete set of base functions, as described in Section 5.3.3.

Benchmarks

For our experiments, we use 10 large combinational MCNC benchmarks and a set of 35 arithmetic circuits, including adders, leading zero detectors, multipliers and majority functions. When the benchmark is a multi-output circuit, we process each output separately, as the interpolation methods can process only a single output at a time. The following subsections describe the results in detail.

5.4.2 Imposing a Single Base Function

For the following experiment, the oracle provides the complete set of base functions G; and, for each $g_i \in G$, we would like to construct a dependency function that has the function g_i in its support set.

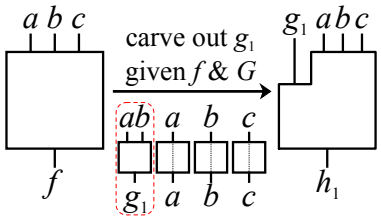


Figure 5.4 – The setup for imposing the base function g_1 using the carving method. The base functions are always expressed as functions of the primary inputs, thus the selected base function is accompanied with the essential identity functions of the target function's primary inputs a, b, and c. After deriving the dependency function h_1, we test if g_1 is used as a primary input.

Table 5.1 – Failure rates of the interpolation methods for a single base function. The given numbers represent percentage of disconnected wanted base functions.

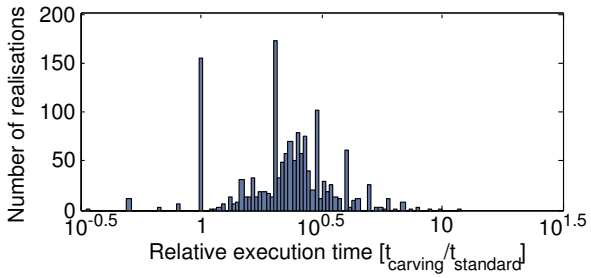
	Standard	Carving
Arithmetic Circuits	55.71%	0.00%
Large MCNC (area-optimised)	59.12%	0.15%
Large MCNC (delay-optimised)	56.40%	0.15%

Experimental setup for imposing a single base function

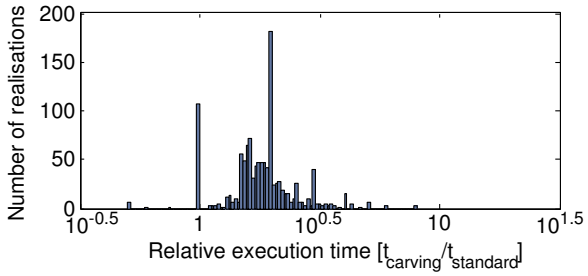
In order to create a dependency function, the target function f should functionally depend on the given set of base functions. Hence, to impose a single base function g_i, we first form a set G_i that contains the function g_i and all identity functions of the primary inputs of the target function. Next, we remove the added identity functions that are auxiliary given the function f and the set G_i. Thus, G_i contains only the identity functions of the primary inputs that are essential for achieving functional dependency and the function g_i. If a removed identity function represents a primary input that is also a primary input of g_i, both the standard and the carving method must use g_i to recreate the target function. Otherwise, if the identity functions of all primary inputs of g_i are given, then several dependency functions exist—some that use the function g_i, and others that reimplement the logic of g_i with the identity functions from G_i. Figure 5.4 shows the setup for imposing the base function g_1 with the carving method.

Failure rates in terms of omitted base functions

In our experiment, when it is possible to return a dependency function that omits the selected base function, we count the number of missed opportunities to use the base function for the two interpolation methods. Table 5.1 summarises the failure rates of the interpolation methods.



(a) Arithmetic benchmarks.



(b) Area-optimised MCNC benchmarks.

Figure 5.5 – The relative execution time to generate a dependency function for a base function set composed of a single base function and essential identity functions.

For the arithmetic circuits, the standard interpolation method fails to use the selected base function in 55.71% of the cases, whereas the carving method always imposes it. For the MCNC benchmarks, when the input structure of the circuits is optimised for area, we observe 59.12% and 0.15% failure rate for the standard method and the carving method, respectively. Similarly, when the input structure is optimised for delay, we obtain 56.40% and 0.15% failure rate, respectively.

Runtime comparison

Regarding the runtime of the two methods, as expected, the carving method takes on average twice the time required by the standard interpolation method, because the dependency function is derived from two interpolants instead of one. Figure 5.5 shows the distribution of the relative execution time for the arithmetic circuits and the area-optimised MCNC benchmarks.

5.4.3 Imposing a Set of Base Functions

Experimental setup for imposing a set of base function

For the following experiment, the oracle provides the complete set of base functions partitioned into subsets, called *layers*, on which the input

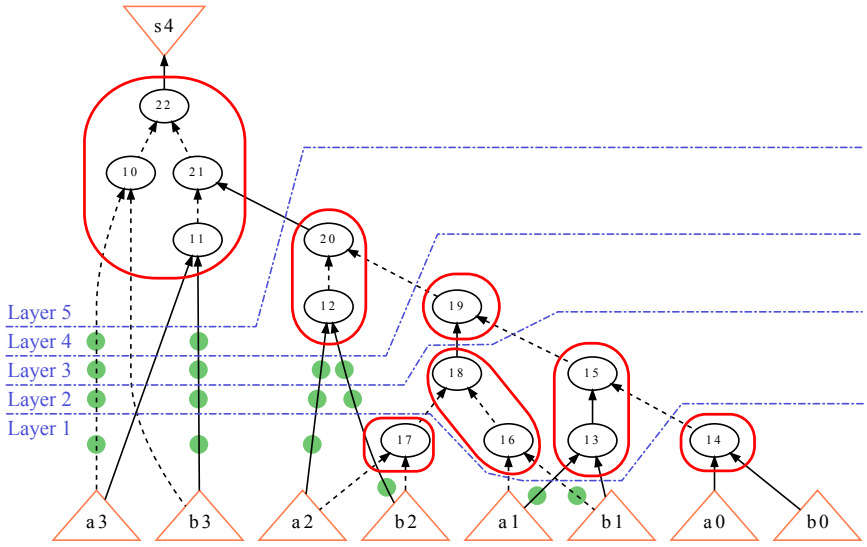
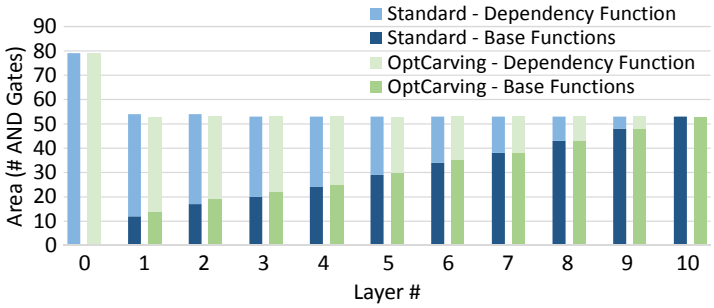


Figure 5.6 – Non-overlapping 3-input cuts and the formed layers for the most significant bit of a 4-bit adder. The rounded sets represent the 3-input cuts whose logic functions are the base functions. The dashed lines divide the cuts in layers. The small ovals show the signals for which an identity function is introduced because they are primary inputs to base functions from a higher layer.

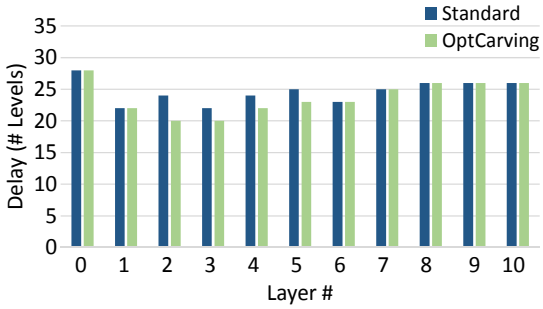
circuit functionally depends. Each layer contains base functions that have only outputs of the previously composed layers as primary inputs. An identity function is introduced for each base function propagated through the layer. Figure 5.6 presents one possible solution for the cuts and layers for the most significant bit of a 4-bit adder. If all the base functions from a layer are essential base functions, then all of them are always used by the standard interpolation method and there is no need to use the carving method. However, if there is at least one auxiliary base function that might be omitted, we consider as failure each layer from which at least one base function was not included in the support set of the built dependency function.

Short description of the compared methods

In Section 5.3.3, we presented two carving methods: the *basic carving method* that imposes all functions from a given set successively, and the *optimised carving method* that first imposes the auxiliary functions and then generates the final dependency function using the standard interpolation method presented in Section 5.2. To compare these with the standard interpolation method, we generate one dependency function by using each of the three methods for each layer received from the oracle. To show the gradual reconstruction of the circuit when using



(a) Area comparison of the standard interpolation and the optimised carving method.



(b) Delay comparison of the standard interpolation and the optimised carving method.

Figure 5.7 – Area and delay of the MSB of a 14-bit adder, reconstructed using 10 layers. The area is expressed in terms of number of AND gates, and the delay is expressed as number of levels of the AIG. For each layer, Figure 5.7a shows the cumulative area of the base functions and the area of the dependency function built with the standard method and with the optimised carving method, respectively. Figure 5.7b compares the delay of the same circuits. In layers 2 and 3, the optimised carving method offers a solution that has lower delay and equal area as the final reference circuit from layer 10, which was obtained by optimising the input circuit from layer 0 in ABC.

layers, Figure 5.7 highlights the area and delay of the recomposed circuit of the MSB of a 14-bit adder after the dependency function is built for each of the 10 layers. For each layer, we verified that the resultant implementation is functionally equivalent to the target function given as input.

Failure rates in terms of layers with at least one omitted base function

The failure rates of the different interpolation methods are summarised in Table 5.2. For the arithmetic circuits, at least one base function from the layer is omitted for 74.59% of the layers, when the dependency function is constructed by the standard interpolation method. In contrast, the basic and the optimised carving method fail to use at least one base function for only 0.26% and 0.07% of the layers, respectively. Although we expected similar results for the MCNC benchmarks, when the input

Table 5.2 – Failure rates of the interpolation methods for a set of base functions.

	Layers with at least one omitted base function			Disconnected base functions among all layers		
		Carving			Carving	
	Standard	Basic	Optim.	Standard	Basic	Optim.
Arithmetic Circuits	74.59%	0.26%	0.07%	64.02%	0.17%	0.04%
Large MCNC (area)	81.82%	83.33%	54.55%	34.00%	26.55%	19.85%
Large MCNC (delay)	92.89%	83.76%	76.14%	77.56%	34.30%	30.77%

structure of the circuits is optimised for area, we obtain 81.82% failure rate for the standard method, whereas for the base and the optimised carving method, the failure rates are 83.33% and 54.55%, respectively. For the same benchmarks, when the input structure is optimised for delay, the failure rates are 92.89% for the standard interpolation method, and 83.76% and 76.14% for the standard and optimised carving method, respectively. These failure rates occur because we consider a failed layer as soon as a single base function is disconnected. For instance, for a given layer, even if the standard method omitted five functions and the carving method has omitted only one, we determine that both methods have failed.

Failure rates in terms of omitted base function among all layers

Therefore, we also analysed the number of disconnected base functions among all layers. The most substantial difference is observed for the large delay-optimised MCNC benchmarks, for which the standard interpolation method omits 77.56%, whereas the standard carving and the optimised carving methods omit only 34.30% and 30.77% of the base functions among all layers, respectively. The results for the other benchmarks are presented in Table 5.2. As Figure 5.8 shows, the carving methods usually have lower failure rates than the standard interpolation method, and fail only for circuits for which the standard interpolation method fails.

Runtime comparison

The basic carving interpolation method builds two interpolants for each base function of the layer, whereas the standard interpolation method constructs one interpolant per layer. Thus, the runtime of the first is, in most cases, significantly higher than the one of the second. Due to the extensive runtime, we fail to execute the algorithm for all the

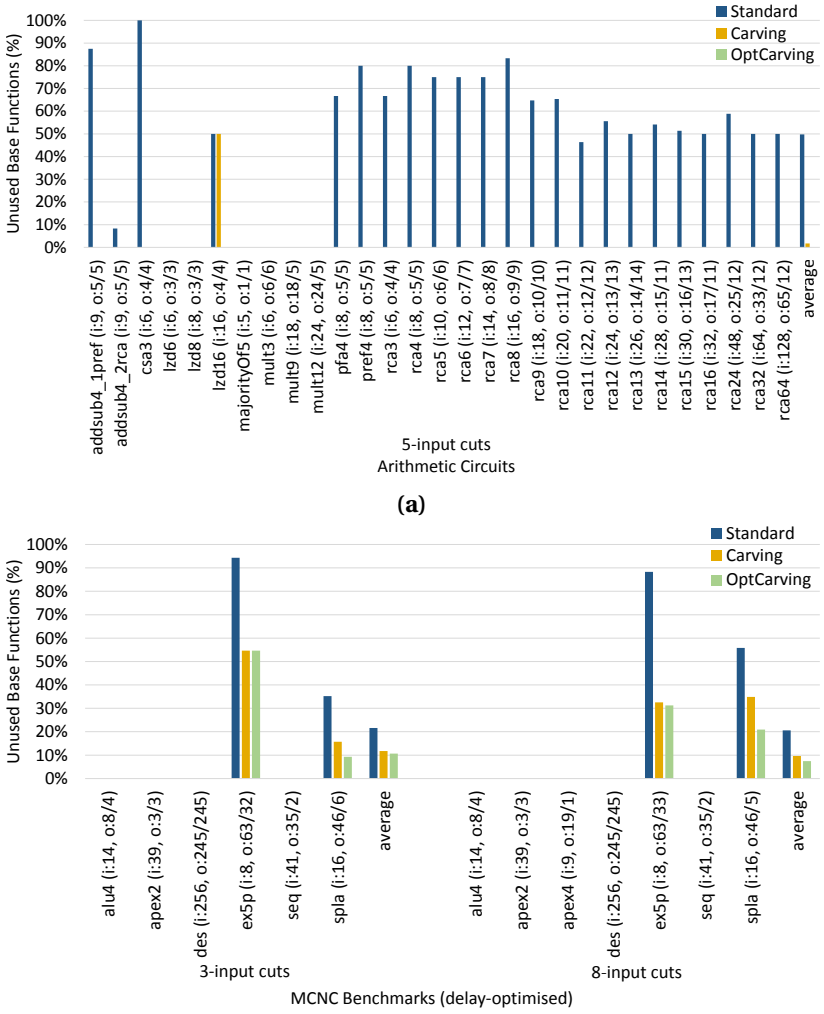
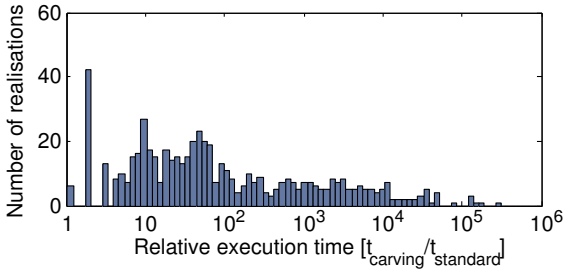
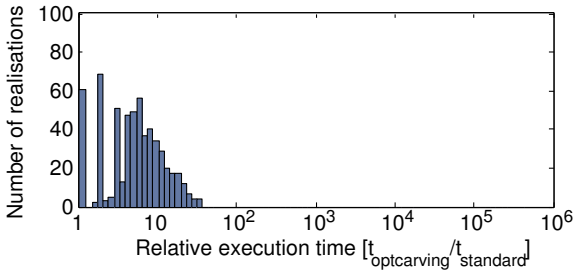


Figure 5.8 – The percentage of disconnected base functions among all layers for the three methods. For the presented benchmarks, the base functions are generated using 5-input cuts for the arithmetic circuits, and using 3- and 8-input cuts for the large combinational delay-optimised MCNC benchmarks. From the MCNC benchmarks, we were able to process at least one output within the given timeout only for the shown benchmark circuits. The formatting of the benchmarks shows their name, the number of primary inputs and outputs, as well as the number of processed outputs for which we report results. For example, the last arithmetic circuit, rca64, has 128 primary inputs and 65 primary outputs, but we processed only the first 12 outputs.

outputs for 28% of the larger MCNC benchmarks. But, for the smaller MCNC benchmarks and the arithmetic circuits, we fail to complete the algorithm for only 2% and 4% of the circuits, respectively. Figure 5.9



(a) The basic carving method is almost always slower than the standard interpolation method, because it computes two interpolants per base function instead of one for the whole set.



(b) The optimised carving method improves over the basic carving method because it computes one interpolant for the essential base functions.

Figure 5.9 – Comparison of the execution time of the interpolation methods when generating a dependency function for a set of base functions. The presented execution time of the carving methods is relative to the execution time of the standard interpolation method.

shows the distribution of the relative execution time for the arithmetic circuits. Although the optimised carving method spends some time dividing the base functions into auxiliary and essential functions, it is generally much faster than the basic carving method. Also, as expected, it succeeds more often in imposing the base functions, as it gives a priority to those that might be omitted.

5.5 Conclusions

Key insights

In this chapter, we have presented a new technique for enabling the reimplementation of an input circuit while imposing a given subcircuit or base function. Our results show that the proposed carving technique is able to successfully include the desired base functions most of the time—it has more than a 99% success rate when forcing a single base function. In comparison, the reference technique, which is based on Craig interpolation, fails far more often. For instance, the success rate of

the standard interpolation method for the large area-optimised MCNC benchmarks barely reaches 40%. Our results also show that our carving technique is moderately slower than the standard interpolation method when forcing a whole set of base functions, but takes only about twice the time when forcing a single base function. This is mostly because the carving technique requires more SAT solver calls and spends more time on reimplementation than the reference technique. To impose a set of base functions, we have also proposed an optimised carving method that represents a hybrid of the basic carving method and the standard interpolation method. This method offers the best failure rate and significantly improves the runtime required for carving, but it is still slower than the standard interpolation method. Consequently, heuristics for global circuit restructuring, as well as synthesis-based ECO algorithms, can benefit from our carving technique when optimising circuits of limited size.

The interpolation method has also been proposed for generating a dependency function during resubstitution. Next, in Chapter 6, we compare a methodology based on (1) a miter similar to the DLN described in Section 5.2 and (2) interpolation to a methodology based on cube enumeration when they are used for resubstitution.

6 Comparison of SAT-Based Algorithms for Resubstitution

Resubstitution in different logic synthesis algorithms

In logic synthesis, *resubstitution* is the problem of replacing a function of a given *target node* from a Boolean network by using a function that depends on other nodes, called *divisors*, without altering the global functionality of the target node [Brayton et al., 1987]. The new implementation of the target node, called a *resubstitution function*, should have better quality compared to the original implementation; for example, in terms of area or delay, which is determined by the optimisation criteria. Many algorithms for multi-level logic optimisation use some kind of resubstitution. For example, it is a part of algorithms for logic optimisation and resynthesis [Mishchenko et al., 2011b, 2017] that can be both technology-independent and technology-dependent, provided that the optimisation is performed before or after the technology mapping, respectively. Also, as we mentioned in Chapter 5, some logic optimisation heuristics for global restructuring [Verma et al., 2009, 2007] build a new implementation for a given circuit by using small single-output circuits as divisors; and some synthesis-based ECO algorithms [Tang et al., 2011; Wu et al., 2010] derive patches, which replace an incorrect subcircuit, as a resubstitution function, but they relax the condition for preserving the global functionality of the network in order to correct the design.

Resubstitution is more general than rewriting

Some fast technology-independent algorithms for *rewriting* minimise an AIG by iteratively resubstituting cuts of the network [Bertacco and Damiani, 1997; Mishchenko et al., 2006a; Yang et al., 2012]. Each cut is replaced with its best implementation, obtained from a pre-computed library of functions. However, in this case, pre-computed functions can be used because the selected function from the library uses exactly the

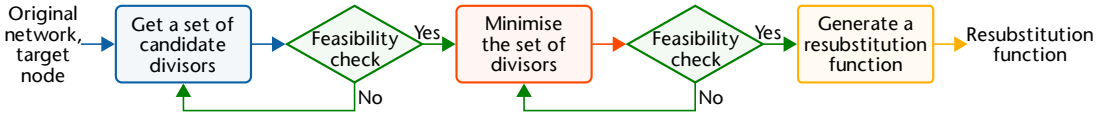


Figure 6.1 – Typical flow of a resubstitution algorithm. The algorithm receives as input the original network and a target node from it, and returns the resubstitution function for the function of the target node. The four main methods are given in boxes with different colours: (1) for obtaining a set of candidate divisors, (2) for checking the feasibility of a given set of divisors, (3) for minimising the set of divisors, and 4) for generating a resubstitution function.

same inputs as the original implementation. In contrast, resubstitution is a more general problem because the resubstitution function can have inputs different than the original one.

Methods required for resubstitution

The standard flow of an algorithm for resubstitution of a given node is illustrated in Figure 6.1. This flow consists of the following steps, each of which can be implemented as a separate method. The flow starts by obtaining *candidate divisors* for the resubstitution of the given node. The candidate divisors are either generated as new nodes in the network, or selected from the existing ones. We have to ensure that the set of candidate divisors is *feasible* for implementing the target node, which means that the function of the target node can be implemented as a function of the given set. However, this initial set of candidates might be large and might include redundant divisors. Therefore, it is minimised and only a subset of it is selected for implementing the resubstitution function. Each divisor from the minimised set of candidate divisors will be an input to the resubstitution function, hence frequently a smaller set leads to a better implementation. Similarly to before, the minimised set has to be feasible for resubstitution. If the set is feasible, then a resubstitution function is generated. Otherwise, a different subset should be selected from the initial set.

Side-to-side comparison of two SAT-based methodologies

In this chapter, we compare two methodologies for implementing all the methods required for resubstitution, except the one for obtaining the initial set of candidate divisors. The algorithms from both compared methodologies receive as input a set of candidate divisors that they minimise while performing a feasibility check, and they generate a resubstitution function at the end. Both algorithms were introduced as building blocks of different SAT-based algorithms for post-mapping

logic optimisation and resynthesis. Additionally, we propose and compare an enhancement for one of them.

Short description of the two compared methodologies

The algorithms from the first methodology [Mishchenko et al., 2011b] perform minimisation and feasibility checking using a *resubstitution miter*, which is a simplified version of the functional dependency miter; and they generate the resubstitution function using the standard interpolation algorithm presented in Chapter 5. However, instead of using simulation for selecting a small feasible subset of divisors, as suggested by Mishchenko et al. [2011; 2006], we propose an efficient algorithm that minimises the set of candidate divisors using a resubstitution miter. The second methodology is based on cube enumeration [Mishchenko et al., 2017], and its algorithm is similar to the one for SAT-based SOP generation presented in Chapter 4. We discussed [Mishchenko et al., 2017] that an algorithm based on a resubstitution miter is faster than an algorithm based on cube enumeration when the set of divisors has more than 10 divisors, but it is slower for smaller sets due to using interpolation. The algorithm based on cube enumeration has an advantage for smaller sets of divisors because it derives the resubstitution function for a node as a by-product of the feasibility check. However, to the best of our knowledge, there is no side-by-side numerical and detailed comparison of them such as the one that this chapter provides.

The remainder of this chapter is organised as follows. In Section 6.1, we describe an algorithm for post-mapping optimisation that is used both as a motivation, and to compare the algorithms in Section 6.3 in which we provide the experimental setup and results. In Section 6.2, we describe the algorithms of the two methodologies for resubstitution. We conclude and present ideas for future work in Section 6.4. The required background information is provided in Section 2.1, Section 2.3, Section 2.4, and Section 2.5.

6.1 Technology Mapping as Motivation

Post-mapping framework for SAT-based logic optimisation

From all applications of resubstitution, we use technology mapping to show the importance of resubstitution and to compare the two methodologies. The methodology based on cube enumeration was introduced as a part of a post-mapping framework for SAT-based logic optimisation with don't-cares that reduces delay and area of the initial map-

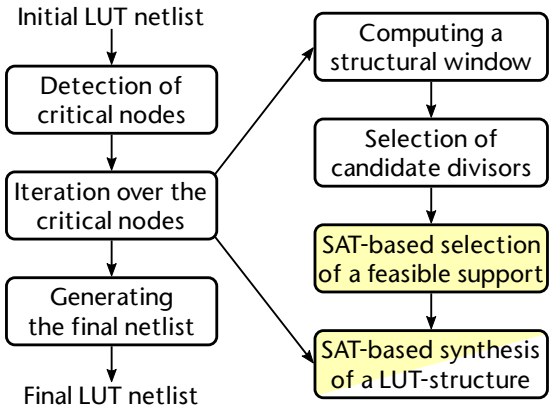


Figure 6.2 – Overview of the framework for SAT-based logic optimisation. The compared SAT-based algorithms are part of the methods in the coloured boxes used iteratively for each critical node [Mishchenko et al., 2017]. They select a minimal feasible set of candidate divisors, and compute the configuration for a k-input LUT when the number of selected divisors is less than or equal to k.

ping [Mishchenko et al., 2017]. Figure 6.2 illustrates the connection between the main methods of the framework. For input, the framework receives an initial mapping as a LUT netlist, from which the area-critical or delay-critical nodes are selected. Then, it iterates through these nodes by prioritising those that can gain most from the optimisation. For each node, it executes the following four methods for optimisation. First, a *structural window* of the node is computed: It contains (1) a fixed number of TFI/TFO levels of logic centered at the node, and (2) all paths from the included TFIs to the included TFOs. The left subcircuit from Figure 6.4 illustrates this window for a given target node n. Next, from the nodes included in the window, a set of nodes is selected for candidate divisors. The set is minimised by selecting a feasible subset of candidate divisors, which would represent the support of the resubstitution function, or it is proved that no feasible subset exists. Last, if the minimised subset is feasible and if it contains at most k divisors, the target node can be reimplemented using one k-input LUT, and its configuration is obtained by generating a resubstitution function. Otherwise, if the number of divisors is larger than k, the resubstitution function is generated as a LUT-structure composed of k-input LUTs that is synthesised using an algorithm based on *quantified Boolean formula (QBF)* solving. Finally, either after iteratively processing all critical nodes, or if a timeout occurs, the final restructured netlist of LUTs is

generated and returned, and it has either equal or better quality than the initial LUT netlist.

The framework can be easily extended to other technologies

This framework is introduced as part of a set of logic synthesis tools for FPGAs, so the initial and resulting mapping are represented as LUT netlist. However, it can be easily extended to other technologies, such as standard cells, technology-independent AIGs, and logic structures composed of known primitives. To achieve this, only the last method from the iterative process, which performs the LUT-structure synthesis, should be replaced with another implementation for the appropriate technology [Mishchenko et al., 2016, 2015].

6.2 SAT-Based Algorithms for Resubstitution

In this section, first, we give the necessary and sufficient condition for the existence of resubstitution that is used by both compared methodologies. Then we describe algorithms based on the two methodologies for resubstitution that are compared in this chapter.

Necessary and sufficient condition for the existence of resubstitution

For the feasibility check, both methodologies use the following theorem for *sets of pairs of functions to be distinguished (SPFD)* [Mishchenko et al., 2006c], which is similar to Theorem 1 for functional dependency. It provides a necessary and sufficient condition for the feasibility of resubstitution.

Theorem 4. Let X be a set of variables $X = \{x_1, \ldots, x_m\}$. The set of divisors $D = \{d_1, d_2, \ldots, d_k\}$ with functions $G = \{g_1(X), g_2(X), \ldots, g_k(X)\}$ is feasible for resubstitution of a target node n with function $f(X)$ if and only if there is no minterm pair (M_1, M_2) such that $f(M_1) \neq f(M_2)$ but $g_i(M_1) = g_i(M_2)$ for all i, where $1 \leq i \leq k$ [Mishchenko et al., 2006c].

In short, a set of candidate divisors is infeasible if it does not distinguish two minterms that a target node distinguishes.

6.2.1 Resubstitution Using a Resubstitution Miter and Interpolation

Link between feasibility of a resubstitution and functional dependency

The similarity between Theorem 1 and Theorem 4 shows that the feasibility of a resubstitution is closely related to the concept of functional dependency presented in Section 2.4. Indeed, the functions of the tar-

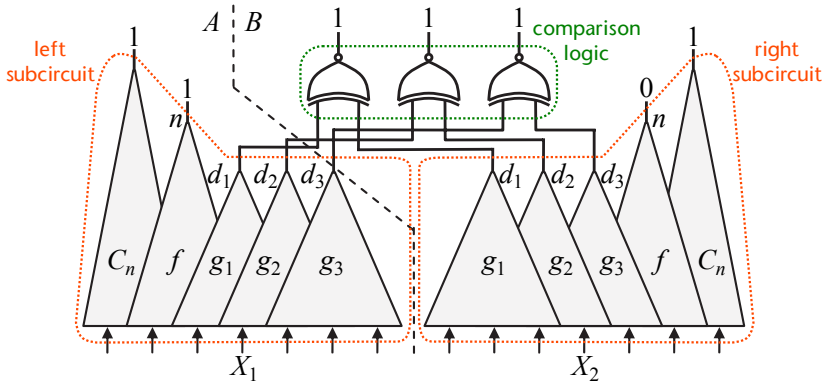


Figure 6.3 – The resubstitution miter. It can be used both for feasibility checking and minimisation of the set of candidate divisors, as well as for computing an interpolant.

get node and divisors, and the resubstitution function in resubstitution represent precisely the target function, base functions, and dependency function, respectively, in functional dependency. Consequently, a version of the miter for checking of functional dependency can be used as a resubstitution miter for feasibility checking, and interpolation can be used to generate the resubstitution function [Mishchenko et al., 2011b] similarly to how they are used in Chapter 5.

Structure of the resubstitution miter

Subsequently, the first compared methodology is based on the resubstitution miter introduced in an integrated SAT-based logic optimisation methodology [Mishchenko et al., 2011b]. In this chapter, we propose a more flexible version of it, which is illustrated on Figure 6.3. It only differs in the comparison logic—instead of using XOR gates connected with an OR gate with output assigned to 0, we propose to use XNOR gates with outputs assigned to 1. For a set with k divisors, the original comparison logic represents the equation

$$(g_{1l} \oplus g_{1r}) + \cdots + (g_{kl} \oplus g_{kr}) = 0,$$

where g_{il} and g_{ir}, for $1 \leq i \leq k$, are the g_i functions from the left and right subcircuit, respectively. If we negate both sides of this equation, we would obtain the proposed version

$$(\overline{g_{1l} \oplus g_{1r}}) \cdot \cdots \cdot (\overline{g_{kl} \oplus g_{kr}}) = 1.$$

Consequently, the function of the proposed miter is the same as the original one, but the independence of the XNOR gates brings flexibility when minimising the set of candidate divisors by using incremental SAT solving with assumptions. Also, this miter has the same functionality as the miter for functional dependency checking from Figure 2.3, but it is simplified because it works only for a single-output function $f(X)$. Similarly to the functional dependency miter, it consists of two copies of the function $f(X)$ for the target node n, and of the functions $g_i(X)$, where $1 \leq i \leq k$, for the divisors d_i. However, additionally, this resubstitution miter considers the care set represented with the logic function $C_n(X)$, which is illustrated in Figure 6.4 and explained in Section 6.2.2. The output of $C_n(X)$ is assumed to be 1 in order to assign X_1 and X_2 only to care-set minterms of n, and to ignore the don't-care ones.

Auxiliary and essential divisors

Similarly to how we defined essential and auxiliary base functions in Section 2.4, we define essential and auxiliary divisors. A divisor $g_i \in G$ is *essential*, if the feasible set G becomes infeasible when g_i is removed from it. Otherwise, g_i is *auxiliary*, and both G and $G \setminus \{g_i\}$ are feasible sets.

Feasibility checking using a resubstitution miter

A SAT solver initialised with a resubstitution miter can check if a set of divisors $D = \{d_1, d_2, \ldots, d_k\}$ is feasible to resubstitute the given function $f(X)$ of the target node n. If the problem is satisfiable, then the set D is infeasible because the satisfying assignment of the variables X_1 and X_2 defines two minterms M_1 and M_2 that are distinguished by the target node n but are not distinguished by any of the divisors d_i. Otherwise, if the problem is UNSAT, the set D is feasible because such pair of minterms does not exist, and the resubstitution is possible.

Fast minimisation based on a resubstitution miter using a single SAT call

For minimising the set of divisors, we consider the following three options. First, when the problem representing the resubstitution miter is UNSAT, the SAT solver returns the set of literals for UNSAT. Thus, the divisors for which the literal is returned define a feasible subset of divisors. However, this feasible subset is not always minimum with respect to the included divisors, hence often redundant divisors can be additionally removed. Moreover, the removal for some divisors can prevent obtaining the global minimum, as illustrated with the Example 6.2.1. The advantage of this algorithm, which in this chapter we call `MITER0`, is that it minimises the initial set with a single SAT call.

Example 6.2.1. Let $X = \{x_1, x_2, x_3, x_4, x_5\}$ be a set of variables. Two different resubstitution functions can be obtained for the target function $f(X) = (x_1 \oplus x_2)(x_3 + x_4)$ with respect to the set of divisors $D = \{d_1, d_2, d_3, d_4\}$ with functions $g_1(X) = x_1 \oplus x_2$, $g_2(X) = x_1 + x_2$, $g_3(X) = x_1 x_2$, and $g_4(X) = x_3 + x_4$. The best implementation with two divisors is $f(X) = g_1(X)g_4(X)$. But, if the divisor d_1 is removed first, then the set is still feasible and the target function can be implemented as $f(X) = g_2(X)\bar{g}_3(X)g_4(X)$.

Minimisation using assumptions and one round of SAT solving

Another option for minimisation is to use incremental SAT solving with assumptions for the *XNOR variables*, which represent the SAT variables of the XNOR gates outputs from the resubstitution miter. A divisor is used if its corresponding XNOR variable is assumed 1; otherwise, it is not used. The simplest iterative solution, which we call `MITER1`, is to try to remove greedily the divisors one by one. Assume that the set of candidate divisors D initially contains k divisors, $D = \{d_1, d_2, \ldots, d_k\}$. In iteration i, where $0 \leq i \leq k-1$, we try to remove the divisor d_{k-i} by assuming the XNOR variables of all divisors to 1, excluding the ones for d_{k-i} and the divisors that are removed in the previous iterations j, where $0 \leq j < i$. The SAT problem with the given assumptions is UNSAT when the divisor can be removed, and we can additionally use the set of assumptions for UNSAT to remove other redundant divisors and to reduce the number of SAT calls. Otherwise, the SAT problem is satisfiable when the divisor is essential. This heuristic performs one round of SAT solving with at most k SAT calls, when the initial set contains k divisors. Unlike `MITER0`, `MITER1` ensures that all divisors in the final minimised set are essential. However, we still might encounter the problem from Example 6.2.1 if the divisor d_1 is removed before the divisors d_2 and d_3.

Efficient minimisation using assumptions and two rounds of SAT solving

Therefore, to obtain a set that is equal or close to the global minimum, we propose the algorithm `MITER2` that performs two rounds of SAT solving. In the first round, when we have k divisors, one SAT call with $k-1$ assumptions is performed for each divisor. In iteration i, we check if the divisor d_{k-i} can be removed, which means that all XNOR variables are assumed to 1 except the one for the divisor d_{k-i}. If the problem is UNSAT, we mark the divisor as auxiliary and we obtain the subset of divisors additionally removed using the set of assumptions for UNSAT. With this round, we retrieve the option that has the highest chance of resulting in the smallest possible subset. Then, in the second round, we consider as removed the subset of divisors that is defined with the

best option from the first round, and we iteratively try to remove the remained divisors that were marked as auxiliary in the first round. This algorithm has linear complexity in the number of candidate divisors because it performs at most two SAT calls for each divisor.

Example 6.2.2. Assume the target node n with function $f(X)$ and the set of divisors D from Example 6.2.1. In the first round we would obtain the following results in the given iterations.

- Iteration 1) The divisor g_1 is auxiliary, and when it is removed the other divisors become essential. Hence, we can remove in total one divisor.
- Iteration 2) The divisor g_2 is auxiliary, and we can remove in total two divisors, g_2 and g_3.
- Iteration 3) The divisor g_3 is auxiliary, and we can remove in total two divisors, g_2 and g_3.
- Iteration 4) The divisor g_4 is essential.

Iterations 2 and 3 lead to the same best result—g_2 and g_3 can be removed simultaneously, and the minimised set would be $D' = \{g_1, g_4\}$. As we do not know if the set is minimal, in the second round we would try to remove g_1: it is the only divisor from D' that was auxiliary in the set D. However, it cannot be removed, because it is an essential divisor in D'.

Generation of the resubstitution function using interpolation

The proposed algorithms for minimisation `MITER1` and `MITER2` provide a minimal set of divisors in which all divisors are essential—if a divisor is removed, then the set becomes infeasible. Thus, there is no need to use carving interpolation to generate the resubstitution function. Instead, we use the standard Craig interpolation, as described in Section 5.2. In this case, to derive the interpolant, we use the proof of unsatisfiability from a SAT solver initialised with the resubstitution miter that has an identical functionality as the DLN from Figure 5.1. In this chapter, for brevity, we call this algorithm `INTER`.

6.2.2 Resubstitution Based on Cube Enumeration

Miter used for cube enumeration

The algorithm based on cube enumeration is proposed as part of the framework for SAT-based logic optimisation [Mishchenko et al., 2017], which is described in Section 6.1. It receives as input the miter from Figure 6.4 [Mishchenko et al., 2011b, 2017]. It consists of three parts: left

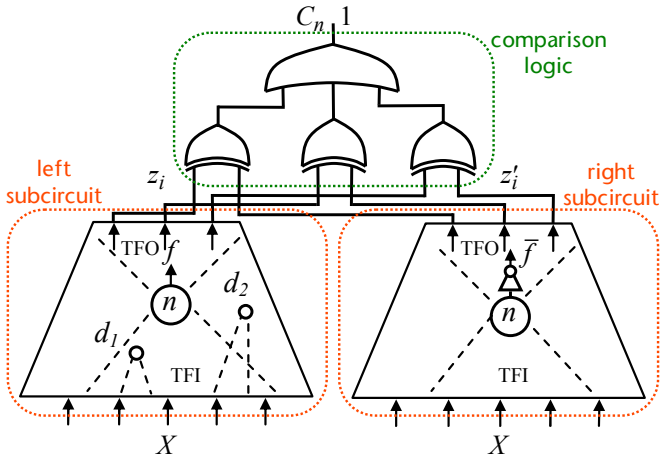


Figure 6.4 – A miter for representing the care set of a target node. In the framework for SAT-based logic optimisation with don't-cares, a SAT solver initialised with this miter computes care-set minterms for the target node n [Mishchenko et al., 2017].

subcircuit, right subcircuit, and comparison logic. The left subcircuit represents the window computed for a target node n, so it includes the function of n, $f(X)$, as well as nodes in its TFI and TFO, and the candidate divisors d_i. The right subcircuit differs from the left one only in the polarity of the function $f(X)$, because an inverter is added at the output of $f(X)$. POs of the left and right subcircuit are $z_i(X)$ and $z'_i(X)$, respectively. The comparison logic with output $C_n(X)$ detects if the POs from at least one pair $(z_i(X), z'_i(X))$ evaluate to a different value for a given assignment of the PIs X.

Care-set minterms can be computed using a SAT solver initialised with the miter

When a SAT solver is initialised with this miter, a variable for each node is introduced. Thus, when the problem is satisfiable, the returned satisfying assignment returns an assignment for each signal of the network, including the PIs and POs of the left and right subcircuits, the divisors, and the outputs $f(X)$. When the output of the miter $C_n(X)$ is assigned 1, the values for the X variables in the returned assignment represent a care-set minterm M_X, because the POs from at least one pair, $(z_i(X), z'_i(X))$, evaluate to the opposite value due to the different polarity of $f(X)$. The minterm M_X is on-set or off-set minterm depending on whether $f(X)$ evaluates to 1 or 0, respectively. Otherwise, if the problem is UNSAT, a care-set minterm does not exist.

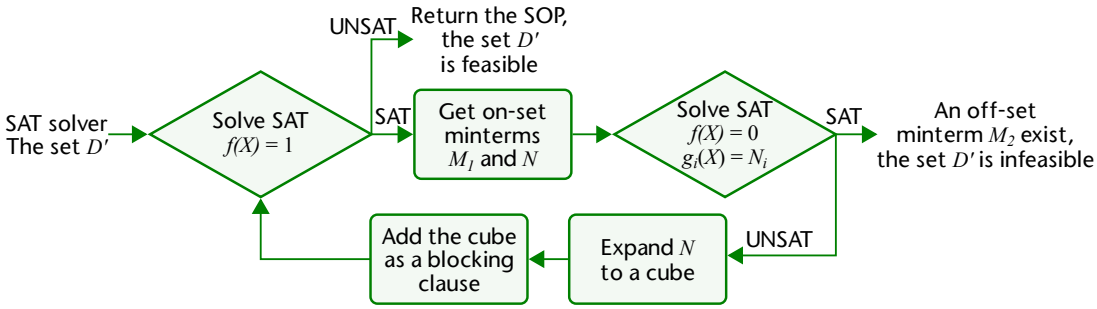


Figure 6.5 – Flow of the algorithm for feasibility check based on cube enumeration. The algorithm proves the set D' as infeasible while computing a partial SOP if a pair of minterms (M_1, M_2), which the divisors from D' cannot distinguish, is found. Otherwise, D' is feasible and a complete SOP representing the function of the target node n as function of the divisors D' is returned.

Minimisation of the set of candidate divisors

The cube enumeration algorithm, which we call CUBEE, iteratively minimises the set of candidate divisors while performing a feasibility check in each iteration. Assume that the set of candidate divisors D initially contains k divisors, $D = \{d_1, d_2, \ldots, d_k\}$. In iteration i, where $0 \leq i \leq k-1$, the divisor d_{k-i} is removed from D, and a feasibility check is run for the minimised set D' by generating an on-set SOP for the function $f(X)$ with the divisors from D' as inputs, as described below. When the set D' is feasible, the feasibility check returns a complete on-set SOP that proves that the divisor d_{k-i} can be removed from D. Otherwise, if the set D' is infeasible, only a partial SOP is computed, and the divisor d_{k-i} remains in the set D. In both cases, the iterations continue and the process is repeated for the next unchecked divisor. For a set D with size k, the process is repeated k times, and k on-set SOPs are generated, some of which are partial SOPs.

Feasibility check of a set of divisors

To generate on-set SOPs, a SAT solver is initialised with the miter from Figure 6.4. Then, by using this SAT solver with incremental solving and assumptions, the SOP is generated. In each iteration, the algorithm for feasibility checking receives as input the initialised SAT solver and the minimised set of candidate divisors D'. It starts by assuming that the variable of the function $f(X)$ in the left subcircuit is 1. As previously described, the SAT solver returns a satisfying assignment for all variables of the SAT problem. The assignment for the variables X define an on-set minterm M_1 for the function f in terms of X. Similarly, the assignment for the variables assigned to the divisors from D' define an on-set minterm N for the function f in terms of the divisors from D'.

Thus, for each bit of N, we have $N_i = g_i(M_1)$, where $1 \leq i \leq k$. Next, to represent the target function in terms of the divisors, the minterm N can be expanded to a cube with the fast non-canonical expansion from Section 4.1.2 with one modification. Instead of using an off-set SAT solver, the variable of the function $f(X)$ in the left subcircuit is assumed to evaluate to 0, and each divisor's variable is assumed to evaluate to its corresponding value from N. If the problem is UNSAT, we can proceed with the fast non-canonical expansion. To ensure generating minterms that are not yet covered, the computed cube is added as a blocking clause to the SAT solver. Yet, unlike the SAT-based SOP generation from Chapter 4 where a function can always be represented using its PIs and the SAT problem is always UNSAT, in this case, the problem is satisfiable if D' is infeasible. In such a case, the SAT solver returns a satisfying assignment that defines an off-set minterm M_2. As defined with Theorem 4, this minterm proves that the set D' is infeasible because the pair of minterms (M_1, M_2) is distinguished by the function $f(X)$ but not by the set of divisors, which in both cases evaluate to N. Cubes are generated either until all minterms are covered, or until it is proved that the set D' is infeasible.

Difference from the SAT-based SOP generation

This SOP generation operates similarly to the SAT-based SOP generation from Chapter 4, with the following differences. Only the on-set SOP is computed, which can invoke longer runtime when the off-set SOP is smaller. Instead of using two SAT solvers to generate the on-set, only one is used by assuming the value of the variable $f(X)$ to 0 and 1, respectively, in order to consider the off-set and on-set of the target node. The miter for initialisation of the SAT solver is different and enables us to use don't-cares because we compute an SOP for a subcircuit instead of a complete circuit, so TFI/TFO nodes can be easily included. The enumerated cubes are not prime and SOPs are redundant, because the minterms are expanded only with the fast non-canonical expansion from Section 4.1.2, and the algorithm for removing redundant cubes from Section 4.1.3 is not used, which decreases the required runtime for SOP generation.

Generation of the resubstitution function

As the feasibility checking that uses cube enumeration builds an SOP when the set of candidate divisors is feasible, the same algorithm `CUBEE` can be used for the generation of the resubstitution function. Actually, when the set of candidate divisors is minimised iteratively, the resubstitution function is obtained as the last-generated complete SOP.

6.3 Experimental Results

In this section, we first introduce the experimental setup. Then, we compare the algorithms from the two methodologies for resubstitution: (1) when they are used for minimisation with feasibility checking of sets of divisors, and (2) for generating a resubstitution function.

6.3.1 Experimental Setup

Short description of the compared algorithm implementations

The framework for post-mapping logic optimisation, which is described in Section 6.1, is available through the command *&mfsd* in ABC [ABC]. This framework includes the algorithm based on cube enumeration `CUBEE` from Section 6.2.2. In order to compare it with the algorithms based on a resubstitution miter from Section 6.2.1, we implemented them in ABC as well. For their implementation, from ABC, we use its integrated incremental SAT solver derived from an early version of MiniSAT [Eén and Sörensson, 2003], which also provides proof of unsatisfiability for UNSAT problems. First, we implemented the algorithms for minimisation and feasibility checking `MITER0`, `MITER1`, and `MITER2`, as described in Section 6.2.1. For `MITER2`, we also provide an alternative implementation `MITER2PP` that uses the SAT solver interfaces for pushing and popping of assumptions. We use two existing implementations of the algorithm `INTER` from ABC; that both derive an interpolant from a proof of unsatisfiability. The first implementation `INTERTT` returns the interpolation function as a truth table but works only for functions with up to 8 inputs. The second implementation `INTERAIG` has no limits on the number of inputs because it returns the interpolation function as an AIG but has a longer runtime compared to the first one.

Benchmarks

For the comparison, we use the set of 18 proprietary industrial benchmarks, which is used also in Chapter 4. These are large benchmarks whose sizes vary between 2.8K and 196.6K AND nodes. For each benchmark, the resubstitution algorithms are called more than 1230 times. We excluded only the benchmark test15 for which they are called only twice.

Logic synthesis flow

For each benchmark, we run the following flow of commands from ABC. After the circuit is read into ABC (command *read*), it is converted into an AIG (command *&get*), and optimised with the *&dc2* command that performs local optimisation. Next, the framework for logic optimisation, in

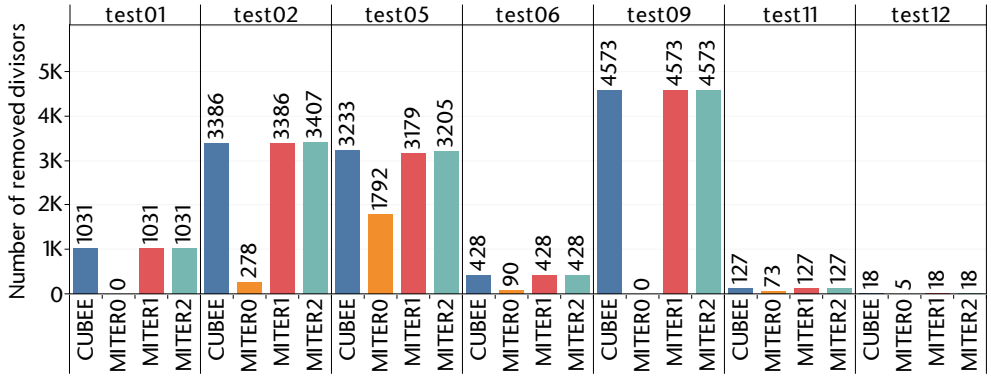


Figure 6.6 – Number of removed divisors by the resubstitution algorithms for minimisation.

which we have integrated all previously mentioned algorithms, is used to create and optimise a LUT mapping (command *&mfsd -K 4 -W 0*). The flag *-K* sets to use 4-input LUTs. The usage of observability don't-cares is disabled with the flag *-W* that sets the number of TFO levels to 0. This enables us to compare fairly the two options while using a simpler structure for the resubstitution miter from Figure 6.3 that excludes the cones C_n. The same satisfiability don't-cares are considered in both cases, because we use the inputs of the window computed by *&mfsd* as inputs of the target node and divisors functions in the resubstitution miter. We call each of the resubstitution algorithms for each critical node selected by *&mfsd*. The window for the target node and the initial set of candidate divisors are used as computed by *&mfsd*. The reported runtime is always an average over three runs of the corresponding algorithm.

Role of `CUBEE` in the command *&mfsd*

The command *&mfsd* uses the algorithm `CUBEE` both for minimisation and feasibility checking of the initial set of divisors, and for generating the resubstitution function when the minimised set of divisors has at most k divisors, where k is the number of inputs of the used LUTs. When the LUT mapping is performed with 4-input LUTs (command *&mfsd -K 4*), on average, `CUBEE` takes 17.9% of the runtime required for the function *&mfsd*; but in some cases it takes up to 30.7% of the total runtime. Reducing its time therefore would lead to a significant runtime reduction of the overall time for logic optimisation.

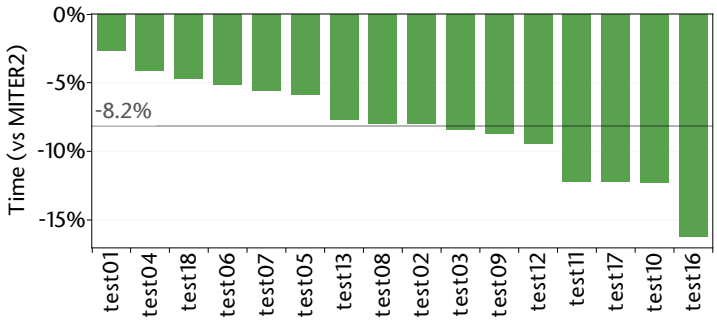


Figure 6.7 – Reduction in runtime achieved by pushing and popping of assumptions. The runtime of the implementation MITER2PP is compared to the runtime of MITER2.

6.3.2 Minimisation and Feasibility Check of a Set of Divisors

Algorithms

For the minimisation and feasibility check of a set of divisors, we compare the algorithms MITER0, MITER1, MITER2, MITER2PP, and CUBEE.

Comparison in terms of quality of results

Regarding the quality of results, we notice that the algorithm MITER0 removes only 17.5% of the divisors removed by the other algorithms because it uses only the set of assumptions for UNSAT that is often a suboptimal set. Regarding the other algorithms, we minimised the sets of divisors only for 7 benchmarks, because the initial divisor sets received from *&mfsd* usually consist mostly of essential divisors and there are only few opportunities for minimisation. As Figure 6.6 shows, the algorithms MITER1, MITER2, and CUBEE removed almost the same number of divisors in total, with the algorithm MITER2 being the most efficient because it tries to select the best option for removal.

Runtime reduction using pushing and popping of assumptions

As the algorithm MITER2 generates the best quality results, we also provide the implementation MITER2PP for it that uses the interfaces for pushing and popping of assumptions. These interfaces enable preserving the internal state of the solver between consecutive SAT calls. As many calls differ in only one assumption, as Figure 6.7 shows, MITER2PP reduces the runtime up to 16.3% compared to MITER2, and achieves an average reduction of 8.2%. A similar reduction can be also obtained for the algorithm MITER1.

Comparison in terms of performance

Regarding the runtime of all algorithms, CUBEE has the longest runtime. Compared to it, the algorithm MITER1 reduces the runtime by 89.7%. As we mentioned previously, it is the least efficient for minimising sets of

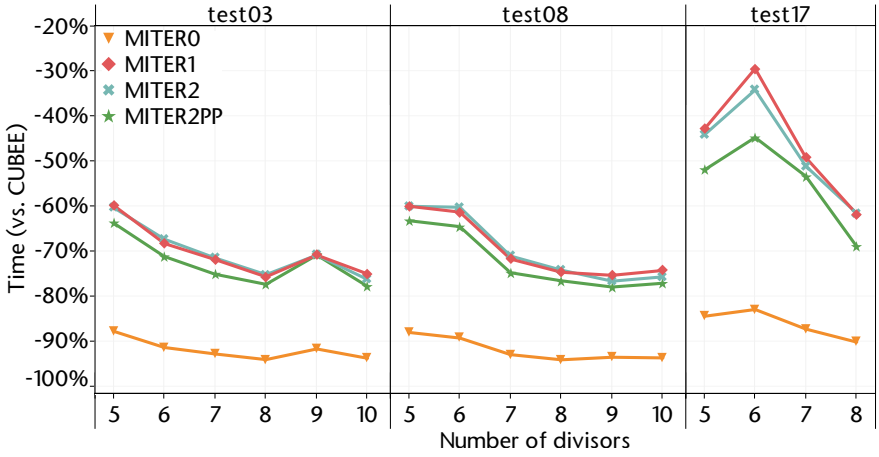
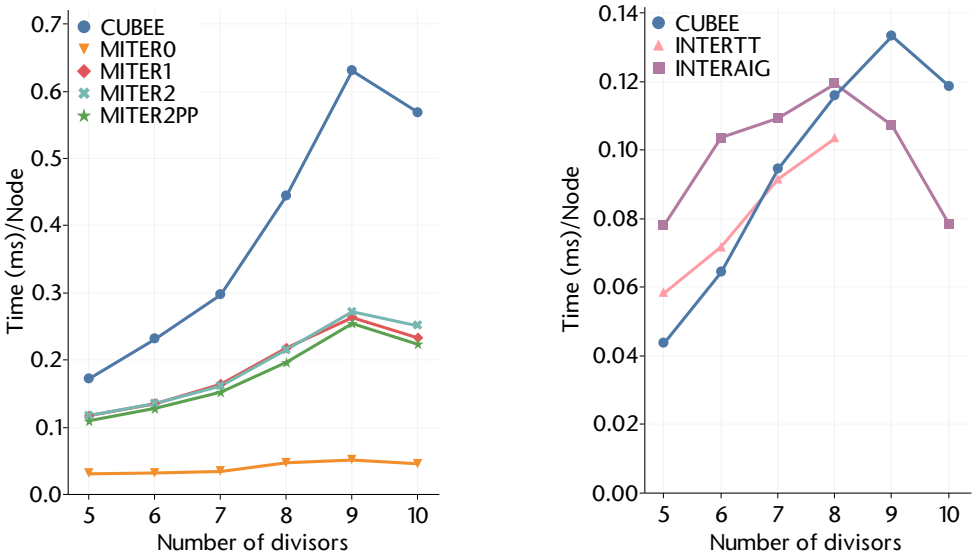


Figure 6.8 – Reduction in runtime for the algorithms based on a resubstitution miter compared to the algorithm based on cube enumeration. We show the three benchmarks with the highest number of calls to the resubstitution algorithms.

divisors, but it is the best option if only feasibility checking is required. Next, compared to CUBEE, the algorithms MITER1 and MITER2 decrease the runtime by 54.8% and 54.4%, respectively. They have similar runtime because, in the first round, both of them require k SAT calls when the size of the initial set of divisors is k. The algorithm MITER2 executes additional SAT calls in the second round only for the remaining auxiliary divisors. Finally, the implementation with the pushing and popping of assumptions MITER2PP reduces the runtime by 58.1%, on average, compared to CUBEE. Figure 6.8 shows the runtime of the algorithms based on a resubstitution miter relative to the runtime of the algorithm CUBEE for three benchmarks with the highest number of calls to the resubstitution algorithms. It compares separately the runtime for sets with a different number of divisors separately, and shows that the runtime reduction of the algorithms based on a resubstitution miter are directly proportional to the number of divisors in the initial set. The runtime of CUBEE increases faster because the number of cubes in the generated SOPs increases exponentially in the worst case; hence, it requires more SAT calls compared to the algorithms based on a resubstitution miter. Similarly, Figure 6.9a illustrates the average runtime per node for the minimisation and feasibility check of sets with different number of divisors.



(a) Minimisation of the set of divisors.

(b) Generation of resubstitution functions.

Figure 6.9 – Comparison of the average runtime per node of the resubstitution algorithms for sets of divisors with different size.

6.3.3 Generation of Resubstitution Function

The resubstitution function is generated regardless of the number of divisors in the minimised set

As in our flow of commands we use 4-input LUTs, the command *&mfsd* requires a resubstitution function only when the final set of divisors has at most four divisors, and it can be implemented with one LUT. Otherwise, to resubstitute the function of a target node, *&mfsd* generates a LUT-structure by using QBF solvers. However, other algorithms, such as the ones for ECO and global reconstructing, might require generating a resubstitution function with more than four inputs. Thus, we compute a resubstitution function with the algorithms `CUBEE`, `INTERTT`, and `INTERAIG`, regardless of the size of the final set of divisors.

Using only one iteration of `CUBEE` to generate a resubstitution function

The algorithm `CUBEE` returns an on-set SOP of the required resubstitution function in the last iteration performed for the minimisation of the set of divisors. However, as Figure 6.9 shows, the minimisation and feasibility check has a longer runtime than the generation of the resubstitution function. Even if we add the runtime required for generating a resubstitution function to the algorithm `MITER2PP`, which has the best performance and quality of results among the algorithms based on a resubstitution miter, it would still have better runtime than `CUBEE`.

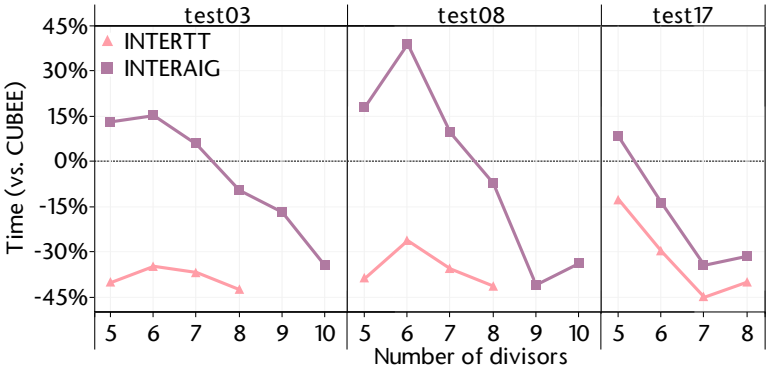


Figure 6.10 – Runtime of the interpolation algorithms for generation of the resubstitution miter relative to the runtime of the algorithm based on cube enumeration. We show the three benchmarks with the highest number of calls to the resubstitution algorithm.

Thus, in this section, we assume that the resubstitution function should always be generated by repeating the process. When comparing the algorithms in this section, to generate the resubstitution function, we run one iteration of CUBEE with the minimised set of divisors as an input.

Comparison in terms of performance

We observe that there is no clear winner among the three algorithms: the best option for generating a resubstitution function depends on the size of the minimised set of divisors. Next, we compare the average runtime of the algorithms among all benchmarks. First, for sets with up to 6 divisors, the algorithm based on cube enumeration CUBEE has the best performance: INTERTT and INTERAIG increase the runtime by 7.1% and 46.6%, respectively. For sets with 7 and 8 divisors, compared to INTERTT, CUBEE and INTERAIG increase the runtime by 15.4% and 21.4%, respectively. Last, when the sets have more than 8 divisors, compared to INTERAIG, CUBEE increases the runtime by 50.2%. The algorithm INTERTT is unavailable in this case because the implementation in ABC that derives the interpolant as a truth table works only when the set of divisors has up to 8 inputs. Figure 6.9b compares the average runtime per node required to generate a resubstitution function for sets with a different number of divisors. Further, Figure 6.8 shows the runtime of the interpolation algorithms relative to the runtime of the algorithm CUBEE for three benchmarks with the highest number of calls to the resubstitution algorithms. It shows that the reductions of the algorithms based on a resubstitution miter are directly proportional to the number of divisors in the initial set. For these benchmarks, the

algorithm INTERTT outperforms CUBEE, regardless of the number of divisors, whereas INTERAIG has a better performance only for larger sets.

6.4 Conclusion

Key insights

Resubstitution is an important part of the algorithms for logic optimisation, but also of ECO and of global restructuring of circuits. In this chapter, we have compared SAT-based algorithms for resubstitution based on two different concepts—(1) based on a resubstitution miter and interpolation, techniques which are also used in Chapter 5, and (2) based on cube enumeration, which is based on the SAT-based SOP generation from Chapter 4.

For minimising and feasibility checking of a set of divisors, the algorithms based on resubstitution miter have better performance compared to the algorithm based on cube enumeration due to the lower number of SAT calls. The one with two rounds of SAT solving has the best trade-off between runtime and quality of results. Yet, the algorithm that uses a resubstitution miter and a single SAT call is the best option when a set of divisors should be checked only for feasibility of resubstitution.

For generating a resubstitution function, determining the best option depends on the number of divisors that will be used for the resubstitution. For sets with up to 6 divisors, the algorithm based on cube enumeration has the best performance. For sets with more than 6 divisors, it is better to use interpolation. Hence, a hybrid approach might work best in some cases.

Possible future improvements

The algorithms based on cube enumeration and interpolation can be further improved in the future. For the one based on cube enumeration, we can compute the on-set and off-set SOPs in parallel, as presented in Chapter 4. This would reduce the runtime when the off-set SOP contains cubes less than the on-set SOP. For interpolation, there are some limitations on the existing implementations that we use from ABC. First, we use the same resubstitution miter both for minimisation of the set of divisors and for generating the resubstitution function. Thus, to generate the interpolant, we can reuse the proof of unsatisfiability from the last SAT call for minimisation. However, the integrated SAT solver

in ABC does not support assumptions and proof logging at the same time. Thus, currently a new SAT solver instance is initialised and run to obtain the proof of unsatisfiability for the interpolant. Second, the function that generates the interpolant as a truth table outperforms the one that generates it as an AIG, but this works only for sets with up to 8 divisors; however, it can be extended to work with larger sets as truth tables have been shown as scalable for functions with up to 16 divisors.

7 Conclusions

SAT solvers are applied across a number of domains in computer science, including EDA. This is mostly due to the continuous improvement in their performance that is primarily enabled by the SAT community. SAT solvers are already used by every EDA vendor [Chelf and Chou, 2008; Malik, 2010] and are included in many academic EDA tools. They are typically the main engine for verifying a design's implementation, but more recently they are also applied for logic synthesis as they have been shown to be the best option so far for many logic synthesis applications.

SAT solvers are already integrated into many EDA tools

However, for certain problems, employing SAT is still challenging for two main reasons. First, the runtime of SAT solvers for a given problem varies significantly, and some hard problems remain unsolved after hours of solving. Second, canonicity is key to some applications in logic synthesis and verification, such as functional equivalence checking, Boolean matching, and caching of subproblems. For a satisfiable problem, SAT solvers can return *any* satisfying assignment; consequently, SAT solvers are often perceived as non-canonical and, as such, inadequate for applications requiring canonicity. Therefore, many logic synthesis and verification applications still rely on canonical BDDs, the fast manipulation of AIGs and other Boolean networks, or use large libraries of known solutions. Yet, in some cases, these traditional methods are incapable of sustaining the continuous growth of design size and complexity, or they have inefficient performance for some practical circuits.

The main restraining reasons for expanding the use of SAT solvers

In this thesis, we have demonstrated that canonicity in SAT-based applications can be attained by using the LEXSAT algorithm, for which

Enabling canonicity in SAT-based applications

we have also proposed a rapid implementation. Methods that typically rely on BDDs for canonicity can now use instead SAT solvers by generating LEXSAT assignments instead of satisfying assignments. Even though generating a LEXSAT assignment requires multiple SAT calls, which can lead to longer runtime, the LEXSAT-based implementations can have better performance compared to conventional techniques. For example, when generating canonical SOPs, our SAT-based method has a better runtime for 5 out of 12 cases compared to the state-of-the-art BDD-based method; and in 6 cases it generates an SOP whereas the BDD-based method fails due to a time limit. Generally, BDDs are canonical by construction, so they lead to canonical results even when canonicity is not required. In contrast, SAT-based algorithms can relax canonicity by simply using regular satisfying assignments in order to achieve better performance. For example, our SAT-based method is 4.3x faster when it generates non-canonical SOPs instead of canonical ones.

Building SAT-based applications with predictable runtime and results

Despite the unpredictable runtime of each call to the SAT solver, we have shown that the SAT-based implementations of some applications can provide intermediate results that enable estimating the final runtime and quality of results. For instance, our SAT-based method for SOPs generates an SOP cube by cube. This progressive nature has several advantages: (1) the algorithm can build partial SOPs for applications that can work with an incomplete functionality of a circuit, and (2) it provides intermediate results that facilitate the estimation of both the additional runtime required to finish the SOP generation and the size of the final SOP. Likewise, when iteratively minimising the set of divisors in resubstitution, we know the number of divisors that we have already checked at any point, hence we can estimate the total runtime for the algorithm. This is another advantage over methods based on BDDs whose termination time and quality of results are unpredictable because the complete BDD has to be built before it can be used.

Introducing new SAT-based algorithms

The new SAT-based algorithms introduced in Chapters 3, 4, and 5, can be used as building blocks in logic synthesis tools. Their initial implementations either have better performance than their state-of-the-art versions, offer new features that are of interest for their target applications, or both. They can therefore ease the development of new logic synthesis algorithms and can help to maintain existing algorithms by upgrading them with the new SAT-based versions of the building blocks.

Applications beyond logic synthesis

Remarkably, the algorithms for LEXSAT and SOP generation can be beneficial to applications from domains beyond logic synthesis, as explained in Chapter 3 and Chapter 4. For example, LEXSAT can be used also for applications from other stages of the EDA flows and for solving specialised SAT problems, such as MAX-SAT, AllSAT, and #SAT, that are practical in various domains; and SOPs are also used in applications such as fuzzy modelling, data compression, and photonic design automation.

7.1 Towards Faster SAT-Based Applications

Dealing with long runtime for hard problems

The long and unpredictable runtime for certain hard problems remains the most challenging concern regarding the use of a SAT solver. In this section, we present some ideas and observations that help in dealing with this problem and that ease the future development of even faster SAT-based logic synthesis applications.

Exploiting features of SAT solvers

In order to have fast SAT-based applications, it is not sufficient to rely on the improvement from the SAT community of the performance of SAT solvers. To make the most of SAT solvers, we must detect and master the SAT solvers' features that are particularly useful for our applications. For example, incremental SAT solving with assumptions is widely used in logic synthesis applications because it enables reusing the initialised SAT solver instance throughout many SAT solver calls. Similarly, many logic synthesis algorithms can benefit from the pushing and popping of assumptions, as proposed and used in this thesis. This feature enables preserving the internal state of the SAT solver between consecutive invocations of the SAT-solving procedure. Through the generation of LEXSAT assignments, expansion of minterms to cubes when generating SOPs, and minimisation of sets of divisors in resubstitution, we have seen that pushing and popping can decrease the runtime for SAT solving when we execute consecutive SAT calls that differ in one or few assumptions. In the same way that LEXSAT enables canonicity, we also need to discover other specialised SAT-solving techniques that bring new features to SAT-based applications.

Creating and exploiting opportunities for parallelisation

Although our algorithms are implemented sequentially, there are many opportunities for parallelisation, which could significantly improve their performance. In particular, for the SAT-based SOP generation we

discuss that (1) the on-set and off-set SOPs can be generated in parallel, and (2) we can parallelise the generation of an SOP for each output. Also, the presented sequential implementation of carving interpolation is on average two times slower than the Craig interpolation, because the carving interpolant is composed out of two Craig interpolants. However, as these two Craig interpolants are independent, the runtime can be improved by computing them in parallel. Lastly, in the proposed resubstitution algorithm with two rounds of SAT solving, the k SAT calls from the first round can be executed as k separate processes. Each process obtains the total number of removed divisors when the corresponding divisor is removed. Many other SAT-based algorithms in logic synthesis have similar opportunities for parallelisation, which can be evaluated in future research.

Efficient encoding and execution of logic synthesis problems as SAT

Finally, we have to find efficient ways to encode the logic synthesis problems as SAT. We can learn some general techniques and approaches from the existing SAT-based applications where SAT solvers are already used efficiently. Also, in order to achieve shorter runtimes for hard problems, it is often better to use several calls to the SAT-solving procedure instead of using one, either by using assumptions or by splitting the hard problem into multiple easier subproblems. For example, we can divide any SAT problem into 2^n independent subproblems by using assumptions for n variables. This is similar to how carving interpolation divides a SAT problem into two subproblems by assigning a variable to 0 and 1, respectively. If at least one subproblem is satisfiable, then it returns a satisfying assignment that proves the satisfiability of the original SAT problem. Otherwise, the evaluation of all subproblems to UNSAT proves that the original problem is UNSAT. This is yet another opportunity to use parallelism as suggested previously, as the 2^n subproblems are independent and can be run in parallel.

7.2 Final Remarks

In this thesis, we have enabled the integration of SAT solvers in applications that require canonicity and introduce novel efficient SAT-based building blocks that can be easily included into logic synthesis tools. The proposed features and ready-to-use algorithms bring logic synthesis, but also EDA, one step closer to a quick and efficient utilisation of SAT solvers with all their appealing features.

However, this is still a thriving research direction with many opportunities for improvement. BDDs have gone through 35 years of continuous in-depth research to gain their current strength. Such dedicated research of exploiting SAT solvers specifically for logic synthesis, combined with the support from the SAT community that improves the SAT solvers performance for any application, would strengthen their role in logic synthesis. This would help to overcome the remaining limitations of SAT-based algorithms. More importantly, it would enable more powerful logic synthesis tools that can produce circuit implementations better than the current ones and that can keep up with the growing size and complexity of designs.

Bibliography

ABC (n.d.). *ABC: A System for Sequential Synthesis and Verification*. http://www.eecs.berkeley.edu/~alanmi/abc/. Berkeley, Calif.: Berkeley Logic Synthesis and Verification Group, n.d.

Akers, Sheldon B. (1978). "Binary decision diagrams". In: *IEEE Transactions on Computers* 27.6 (June 1978), pp. 509–16.

Alcorn, Paul (2016). "Intel Xeon E5-2600 v4 Broadwell-EP review". In: *Tom's Hardware* (Mar. 2016). Accessed April 14, 2017, http://www.tomshardware.com/reviews/intel-xeon-e5-2600-v4-broadwell-ep,4514-2.html.

Aloul, Fadi A., Karem A. Sakallah, and Igor L. Markov (2006). "Efficient symmetry breaking for Boolean satisfiability". In: *IEEE Transactions on Computers* 55.5 (2006), pp. 549–58.

Altera (2016). *Stratix 10 GX/SX Device Overview*. Accessed April 14, 2017, https://www.altera.com/en_US/pdfs/literature/hb/stratix-10/s10-overview.pdf. Intel Corporation. Oct. 2016.

Amarù, Luca, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli (2014a). "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization". In: *Proceedings of the 51st Design Automation Conference*. San Francisco, Calif., June 2014, pp. 1–6.

Amarù, Luca, Pierre-Emmanuel Gaillardon, Andreas Burg, and Giovanni De Micheli (2014b). "Data compression via logic synthesis". In: *Proceedings of the Asia and South Pacific Design Automation Conference*. Yokohama, Japan, Jan. 2014, pp. 628–33.

Amarù, Luca, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli (2013). "Biconditional BDD: A novel canonical BDD for logic synthesis targeting XOR-rich circuits". In: *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. Grenoble, France, Sept. 2013, pp. 1014–17.

Balyo, Tomas, Marijn J. H. Heule, and Matti Järvisalo (2017). "SAT competition 2016: Recent developments". In: *Proceedings of the 31st AAAI Conference on Artificial Intelligence*. San Francisco, Calif., Feb. 2017, pp. 5061–63.

Belle, Vaishak (2016). "Satisfiability and model counting in open universes". In: *Proceedings of the 2016 AAAI Workshop on Beyond NP*. Phoenix, Az., Feb. 2016.

BenchIBM (n.d.). *The EPFL Combinational Benchmark Suite, "Multi-output PLA benchmarks"*. http://lsi.epfl.ch/benchmarks. n.d.

Bertacco, Valeria and Maurizio Damiani (1997). "The disjunctive decomposition of logic functions". In: *Proceedings of the International Conference on Computer Aided Design*. San Jose, Calif., Nov. 1997, pp. 78–82.

Berg, Jeremias, Antti Hyttinen, and Matti Järvisalo (2015). "Applications of MaxSAT in data analysis". In: *Proceedings of the 6th Pragmatics of SAT Workshop*. Austin, Tex., Sept. 2015.

Betz, Vaughn, Jonathan Rose, and Alexander Marquardt (1999). *Architecture and CAD for deep-submicron FPGAs*. Boston, Mass.: Kluwer Academic, 1999.

Biere, Armin, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu (1999). "Symbolic model checking using SAT procedures instead of BDDs". In: *Proceedings of the 36th Design Automation Conference*. New Orleans, La., June 1999, pp. 317–20.

Björk, Magnus (2009). "Successful SAT encoding techniques". In: *Journal on Satisfiability, Boolean Modeling and Computation* (July 2009).

Bollig, Beate and Ingo Wegener (1996). "Improving the variable ordering of OBDDs is NP-complete". In: *IEEE Transactions on Computers* 45.9 (Sept. 1996), pp. 993–1002.

Brayton, Robert K., Gary D. Hachtel, and Alberto L. Sangiovanni-Vincentelli (1990). "Multilevel logic synthesis". In: *Proceedings of the IEEE* 78.2 (Feb. 1990), pp. 264–300.

Brayton, Robert K., Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni Vincentelli (1984). *Logic Minimization Algorithms for VLSI Synthesis*. Boston, Mass.: Kluwer Academic, 1984.

Brayton, Robert K. and Alan Mishchenko (2010). "ABC: An academic industrial-strength verification tool". In: *Proceedings of the International Conference on Computer Aided Verification*. Vol. 6174. Lecture Notes in Computer Science. Springer, July 2010, pp. 24–40.

Brand, Daniel (1993). "Verification of large synthesized designs". In: *Proceedings of the International Conference on Computer Aided Design*. Santa Clara, Calif., Nov. 1993, pp. 534–37.

Brayton, Robert K., Richard L. Rudell, Alberto L. Sangiovanni-Vincentelli, and Albert R. Wang (1987). "MIS: A multiple-level logic optimization system". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 6.6 (1987), pp. 1062–81.

Bryant, Randal E. (1986). "Graph-based algorithms for Boolean function manipulation". In: *IEEE Transactions on Computers* C-35.8 (Aug. 1986), pp. 677–91.

Bryant, Randal E. (1991). "On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication". In: *IEEE Transactions on Computers* 40.2 (1991), pp. 205–13.

Burchard, Jan, Tobias Schubert, and Bernd Becker (2015). "Laissez-Faire caching for parallel #sat solving". In: *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*. Austin, Tex., Sept. 2015, pp. 46–61.

Chang, Kai-Hui, Valeria Bertacco, Igor L. Markov, and Alan Mishchenko (2010). "Logic synthesis and circuit customization using extensive external don't-cares". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 15.3 (May 2010), 26:1–26:24.

Chelf, Ben and Andy Chou (2008). *The Next Generation of Static Analysis: Boolean Satisfiability and Path Simulation—A Perfect Match*. White paper. San Francisco, Calif.: Coverity, Inc., Mar. 2008.

Chen, Chih-Ang and Sandeep K. Gupta (1996). "A satisfiability-based test generator for path delay faults in combinational circuts". In: *Proceedings of the 33rd Design Automation Conference*. Las Vegas, Nev., June 1996, pp. 209–14.

Chen, Yibin, Sean Safarpour, Andreas G. Veneris, and João P. Marques Silva (2009). "Spatial and temporal design debug using partial MaxSAT". In: *Proceedings of the 19th ACM Great Lakes Symposium on VLSI*. Boston, Mass., May 2009, pp. 345–50.

Claessen, Koen, Niklas Eén, Mary Sheeran, Niklas Sörensson, Alexey Voronov, and Knut Åkesson (2009). "SAT-solving in practice, with a tutorial example from supervisory control". In: *Discrete Event Dynamic Systems* 19.4 (2009), pp. 495–524.

Condrat, Christopher, Priyank Kalla, and Steve Blair (2011). "Logic synthesis for integrated optics". In: *Proceedings of the 21st ACM Great Lakes Symposium on VLSI*. Lausanne, Switzerland, May 2011, pp. 13–18.

Cong, Jason and Yuzheng Ding (1994). "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.1 (Jan. 1994), pp. 1–12.

Cook, Stephen A. (1971). "The complexity of theorem-proving procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. Shaker Heights, Ohio: ACM, 1971, pp. 151–58.

Coudert, Olivier (1994). "Two-level logic minimization: An overview". In: *Integration, the VLSI journal* 17.2 (Oct. 1994), pp. 97–140.

Coudert, Olivier, Jean Christophe Madre, and Henri Fraisse (1993a). "A new viewpoint on two-level logic minimization". In: *Proceedings of the 30th Design Automation Conference*. Dallas, Tex., June 1993, pp. 625–30.

Coudert, Olivier, Jean Christophe Madre, Henri Fraisse, and Herve Touati (1993b). "Implicit prime cover computation: An overview". In: *Proceedings of the Synthesis And Simulation Meeting and International Interchange*. Nara, Japan, Oct. 1993.

Courtland, Rachel (2015). "Gordon Moore: The man whose name means progress, an interview". In: *IEEE Spectrum* (Mar. 2015). Accessed April 14, 2017, http://spectrum.ieee.org/computing/hardware/gordon-moore-the-man-whose-name-means-progress.

Craig, William (1957). "Linear reasoning. A new form of the Herbrand-Gentzen theorem". In: *The Journal of Symbolic Logic* 22.3 (Sept. 1957), pp. 250–68.

De Micheli, Giovanni (1991). "Technology mapping of digital circuits". In: *Proceedings of the 5th Annual European Computer Conference*. May 1991, pp. 580–86.

De Micheli, Giovanni (1994). *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

Devadas, Srinivas (1989). "Optimal layout via Boolean satisfiability". In: *Proceedings of the International Conference on Computer Aided Design*. Nov. 1989, pp. 294–97.

Eén, Niklas and Niklas Sörensson (2003). "An extensible SAT-solver". In: *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*. Vol. 2919. Lecture Notes in Computer Science. Springer, May 2003, pp. 502–18.

Eén, Niklas, Alan Mishchenko, and Nina Amla (2010). "A single-instance incremental SAT formulation of proof- and counterexample-based abstraction". In: *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design*. Lugano, Switzerland, Oct. 2010, pp. 181–88.

Fraisse, Henri, Abhishek Joshi, Dinesh Gaitonde, and Alireza Kaviani (2016). "Boolean satisfiability-based routing and its application to Xilinx UltraScale clock network". In: *Proceedings of the 24th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, Calif., Feb. 2016, pp. 74–79.

Fujita, Masahiro and Alan Mishchenko (2014). "Efficient SAT-based ATPG techniques for all multiple stuck-at faults". In: *Proceedings of the International Test Conference*. Seattle, Wash., Oct. 2014, pp. 1–10.

Fujita, Masahiro, Naoki Taguchi, Kentaro Iwata, and Alan Mishchenko (2015). "Incremental ATPG methods for multiple faults under multiple fault models". In: *Proceedings of the 16th International Symposium on Quality Electronic Design*. Santa Clara, Calif., Mar. 2015, pp. 177–180.

Ghosh, Abhijit, Srinivas Devadas, and A. Richard Newton (1991). "Test generation and verification for highly sequential circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 10.5 (May 1991), pp. 652–67.

Gobi, Adam F. and Witold Pedrycz (2007). "Fuzzy modelling through logic optimization". In: *International Journal of Approximate Reasoning* 45.3 (Aug. 2007), pp. 488–510.

Goldberg, Evguenii I., Yuji Kukimoto, and Robert K. Brayton (1997). "Canonical TBDD's and their application to combinational verification". In: *Proceedings of the 6th International Workshop on Logic and Synthesis*. Tahoe City, CA, May 1997.

Goldberg, Evguenii I., Mukul R. Prasad, and Robert K. Brayton (2001). "Using SAT for combinational equivalence checking". In: *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. Munich, Mar. 2001, pp. 114–121.

Goldberg, Evguenii and Yakov Novikov (2002). "BerkMin: A fast and robust SAT-solver". In: *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. Paris, Mar. 2002, pp. 142–49.

Gomes, Carla P., Henry Kautz, Ashish Sabharwal, and Bart Selman (2008). "Satisfiability solvers". In: *Handbook of Knowledge Representation*. Ed. by Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter. Vol. 3. Foundations of Artificial Intelligence. Elsevier, Jan. 2008, pp. 89–134.

Gregory, David, Karen Bartlett, Aart de Geus, and Gary Hachtel (1986). "SOCRATES: A system for automatically synthesizing and optimizing combinational logic". In: *Proceedings of the 23rd ACM/IEEE Design Automation Conference*. Las Vegas, Nev., June 1986, pp. 79–85.

Gu, Jun, Paul W. Purdom, John Franco, and Benjamin W. Wah (1996). "Algorithms for the satisfiability (SAT) problem: A survey". In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. Piscataway, N.J., Mar. 1996, pp. 19–152.

Hellerman, Leo (1963). "A catalog of three-variable Or-inverter and And-inverter logical circuits". In: *IEEE Transactions on Electronic Computers* 12 (1963), pp. 198–223.

Huang, Zheng, Lingli Wang, Yakov Nasikovskiy, and Alan Mishchenko (2013). "Fast Boolean matching based on NPN classification". In: *Proceedings of the 2013 International Conference on Field Programmable Technology*. Kyoto, Dec. 2013, pp. 310–13.

Jabbour, Saïd, Lakhdar Sais, and Yakoub Salhi (2013). "Boolean satisfiability for sequence mining". In: *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management*. San Francisco, Calif., Oct. 2013, pp. 649–58.

Järvisalo, Matti, Daniel Le Berre, Olivier Roussel, and Laurent Simon (2012). "The international SAT solver competitions". In: *AI Magazine* 33.1 (2012), pp. 89–94.

Jiang, Jie-Hong Roland, Chih-Chun Lee, Alan Mishchenko, and Chung-Yang (Ric) Huang (2010). "To SAT or not to SAT: Scalable exploration of functional dependency". In: *IEEE Transactions on Computers* C-59.4 (Apr. 2010), pp. 457–67.

Jiang, Jie-Hong R. and Robert K. Brayton (2004). "Functional dependency for verification reduction". In: *Proceedings of the International Conference on Computer Aided Verification*. Boston, Mass., July 2004, pp. 268–80.

Karnaugh, Maurice (1953). "The map method for synthesis of combinational logic circuits". In: *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics* 72.5 (Nov. 1953), pp. 593–99.

Knuth, Donald E. (2009). *Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Vol. 4. The Art of Computer Programming. Addison-Wesley, Mar. 2009.

Knuth, Donald E. (2015). *Fascicle 6: Satisfiability*. Vol. 19. The Art of Computer Programming. Reading, Mass.: Addison-Wesley, Dec. 2015.

Kravets, Victor N. (2015). "Application of a key-value paradigm to logic factoring". In: *Proceedings of the IEEE* 103.11 (Nov. 2015), pp. 2076–92.

Kuehlmann, Andreas, Viresh Paruthi, Florian Krohm, and Malay K. Ganai (2002). "Robust Boolean reasoning for equivalence checking and functional property verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21.12 (2002), pp. 1377–94.

Larrabee, Tracy (1990). "Efficient Generation of Test Patterns Using Boolean Satisfiability". Ph.D. Thesis. Stanford, Calif.: Department of Computer Science, Stanford University, Feb. 1990.

Le, Bao, Dipanjan Sengupta, and Andreas G. Veneris (2013). "Reviving erroneous stability-based clock-gating using partial Max-SAT". In: *Proceedings of the 18th Asia and South Pacific Design Automation Conference*. Yokohama, Japan, Jan. 2013, pp. 717–22.

Lee, Chin-Chun, Jie-Hong R. Jiang, Chung-Yang Huang, and Alan Mishchenko (2007). "Scalable exploration of functional dependency by interpolation and incremental SAT solving". In: *Proceedings of the International Conference on Computer Aided Design*. San Jose, Calif., Nov. 2007, pp. 227–33.

Lee, Ruei-Rung, Jie-Hong R. Jiang, and Wei-Lun Hung (2008). "Bi-decomposing large Boolean functions via interpolation and satisfiability solving". In: *Proceedings of the 45th Design Automation Conference*. Anaheim, Calif., June 2008, pp. 636–41.

Lee, Chester Chi Yuan (1959). "Representation of switching circuits by binary-decision programs". In: *The Bell System Technical Journal* 38.4 (July 1959), pp. 985–99.

Lin, Hsuan-Po, Jie-Hong R. Jiang, and Ruei-Rung Lee (2008). "To SAT or not to SAT: Ashenhurst decomposition in a large scale". In: *Proceedings of the International Conference on Computer Aided Design*. San Jose, Calif., Nov. 2008, pp. 32–37.

Lin, Chen-Hsuan, Chun-Yao Wang, and Yung-Chih Chen (2009). "Dependent-latch identification in reachable state space". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* CAD-28.8 (Aug. 2009), pp. 1113–26.

Ling, Andrew C., Deshanand P. Singh, and Stephen Dean Brown (2005). "FPGA technology mapping: A study of optimality". In: *Proceedings of the 42nd Design Automation Conference*. San Diego, Calif., June 2005, pp. 427–32.

Malik, Sharad (2010). "SAT Solvers: A Condensed History". http://www.cs.princeton.edu/courses/archive/spring10/cos598D/SharadMalikCOS598d.pdf. Mar. 2010.

Marques-Silva, João (2008). "Practical applications of Boolean satisfiability". In: *Proceedings of the 9th International Workshop on Discrete Event Systems*. Goteborg, Sweden, May 2008, pp. 74–80.

Marques-Silva, João P. (2007). "Satisfiability-Based Model Checking Algorithms". http://pst.istc.cnr.it/RCRA07/articoli/jpms-rcra07-slides.pdf. Rome, July 2007.

Marques-Silva, João P. and Karem A. Sakallah (1999). "GRASP: A search algorithm for propositional satisfiability". In: *IEEE Transactions on Computers* 48.5 (1999), pp. 506–21.

Marques-Silva, João P. and Karem A. Sakallah (2000). "Boolean satisfiability in electronic design automation". In: *Proceedings of the 37th Design Automation Conference*. Los Angeles, Calif., June 2000, pp. 675–80.

Marques-Silva, João, Josep Argelich, Ana Graça, and Inês Lynce (2011). "Boolean lexicographic optimization: Algorithms & applications". In: *Annals of Mathematics and Artificial Intelligence* 62.3 (May 2011), pp. 317–43.

McCluskey, Edward J. (1956). "Minimization of Boolean functions". In: *Bell System Tech. Journal* 35.6 (Nov. 1956), pp. 1417–44.

McMillan, Kenneth L. (2003). "Interpolation and SAT-based model checking". In: *Proceedings of the International Conference on Computer Aided Verification*. Vol. 2725. Lecture Notes in Computer Science. Springer, July 2003, pp. 1–13.

Minato, Shin-ichi (1992). "Fast generation of irredundant sum-of-products forms from binary decision diagrams". In: *Proceedings of Synthesis And Simulation Meeting and International Interchange*. Kobe, Japan, Apr. 1992, pp. 64–73.

Mishchenko, Alan and Robert K. Brayton (2005). "SAT-based complete don't-care computation for network optimization". In: *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. Munich, Mar. 2005, pp. 412–17.

Mishchenko, Alan (2014). *An Introduction to Zero-Suppressed Binary Decision Diagrams*. Technical Report. Berkeley, Calif.: University of California, Berkeley, Feb. 2014.

Mishchenko, Alan, Robert K. Brayton, Stephen Jang, and Victor N. Kravets (2011a). "Delay optimization using SOP balancing". In: *Proceedings of the International Conference on Computer Aided Design*. San Jose, Calif., Nov. 2011, pp. 375–82.

Mishchenko, Alan, Robert K. Brayton, Jie-Hong R. Jiang, and Stephen Jang (2011b). "Scalable don't-care-based logic optimization and resynthesis". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 4.4 (Dec. 2011), 34:1–34:23.

Mishchenko, Alan, Robert K. Brayton, Ana Petkovska, and Mathias Soeken (2017). "SAT-based optimization with don't-cares revisited". In: *Proceedings of the 26th International Workshop on Logic and Synthesis*. Austin, Tex., June 2017.

Mishchenko, Alan, Robert Brayton, Thierry Besson, Sriram Govindarajan, Harm Arts, and Paul van Besouw (2016). "Versatile SAT-based remapping for standard cells". In: *Proceedings of the 25th International Workshop on Logic and Synthesis*. Austin, Tex., June 2016.

Mishchenko, Alan, Robert Brayton, Wenyi Feng, and Jonathan W. Greene (2015). "Technology mapping into general programmable cells". In: *Proceedings of the 23rd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, Calif., Feb. 2015, pp. 70–73.

Mishchenko, Alan, Satrajit Chatterjee, and Robert K. Brayton (2006a). "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis". In: *Proceedings of the 43rd Design Automation Conference*. San Francisco, Calif., July 2006, pp. 532–35.

Mishchenko, Alan, Satrajit Chatterjee, Robert K. Brayton, and Niklas Eén (2006b). "Improvements to combinational equivalence checking". In: *Proceedings of the International Conference on Computer Aided Design*. San Jose, Calif., Nov. 2006, pp. 836–43.

Mishchenko, Alan, Jin S. Zhang, Subarnarekha Sinha, Jerry R. Burch, Robert K. Brayton, and Malgorzata Chrzanowska-Jeske (2006c). "Using simulation and satisfiability to compute flexibilities in Boolean networks". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.5 (May 2006), pp. 743–55.

Moore, Gordon E. (1975). "Progress in digital integrated electronics". In: *International Electron Devices Meetings, Technical Digest* (1975), pp. 11–13.

Morgado, António and João P. Marques-Silva (2005). "Good learning and implicit model enumeration". In: *Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence*. Hong Kong, Nov. 2005, pp. 131–36.

Moskewicz, Matthew W., Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik (2001). "Chaff: Engineering an efficient SAT solver". In: *Proceedings of the 38th Design Automation Conference*. Las Vegas, Nev., June 2001, pp. 530–35.

Nadel, Alexander (2011). "Generating diverse solutions in SAT". In: *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*. Ann Arbor, Mich., June 2011, pp. 287–301.

Nadel, Alexander and Vadim Ryvchin (2016). "Bit-vector optimization". In: *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Eindhoven, The Netherlands, Apr. 2016, pp. 851–67.

Nam, Gi-Joon, Karem A. Sakallah, and Rob A. Rutenbar (1999). "Satisfiability-based layout revisited: Detailed routing of complex FPGAs via search-based Boolean SAT". In: *Proceedings of the 7th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, Calif., Feb. 1999, pp. 167–75.

Oh, Yoonna, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, and Igor L. Markov (2004). "AMUSE: A minimally-unsatisfiable subformula extractor". In: *Proceedings of the 41st Design Automation Conference*. San Diego, Calif., June 2004, pp. 518–23.

Petkovska, Ana, Mathias Soeken, Giovanni De Micheli, Paolo Ienne, and Alan Mishchenko (2016a). "Fast hierarchical NPN classification". In: *Proceedings of the 26th International Conference on Field-Programmable Logic and Applications*. Lausanne, Switzerland, Aug. 2016, pp. 1–4.

Petkovska, Ana, Alan Mishchenko, Mathias Soeken, Giovanni De Micheli, Robert Brayton, and Paolo Ienne (2016b). "Fast generation of lexicographic satisfiable assignments: Enabling canonicity in SAT-based applications". In: *Proceedings of the International Conference on Computer Aided Design*. Austin, Tex., Nov. 2016, pp. 1–8.

Petkovska, Ana, David Novo, Alan Mishchenko, and Paolo Ienne (2014). "Constrained interpolation for guided logic synthesis". In: *Proceedings of the International Conference on Computer Aided Design*. San Jose, Calif., Nov. 2014.

Petkovska, Ana, Alan Mishchenko, David Novo, Muhsen Owaida, and Paolo Ienne (2017). "Progressive generation of canonical irredundant sums of products using a SAT solver". In: *Advanced Logic Synthesis*. Ed. by Rolf Drechsler and Andre Reis. To appear. Springer, 2017.

Pudlák, Pavel (1997). "Lower bounds for resolution and cutting plane proofs and monotone computations". In: *The Journal of Symbolic Logic* 62.3 (Sept. 1997), pp. 981–98.

Quine, Willard V. (1952). "The problem of simplifying truth functions". In: *The American Mathematical Monthly* 59.8 (1952), pp. 521–31.

Rajski, Janusz and Jagadeesh Vasudevamurthy (1992). "The testability-preserving concurrent decomposition and factorization Boolean expressions". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 11.6 (June 1992), pp. 778–93.

Ray, Sayak, Alan Mishchenko, Niklas Eén, Robert K. Brayton, Stephen Jang, and Chao Chen (2012). "Mapping into LUT structures". In: *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. Dresden, Mar. 2012, pp. 1579–84.

Rice, Michael and Sanjay Kulhari (2008). *A Survey of Static Variable Ordering Heuristics for Efficient BDD/MDD Construction*. Technical Report. Riverside, Calif.: University of California, Riverside, 2008.

Rubenstein, Roy (2016). "Altera's 30 billion transistor FPGA". In: *Gazettabyte* (June 2016). Accessed April 14, 2017, http://www.gazettabyte.com/home/2015/6/28/alteras-30-billion-transistor-fpga.html.

Rudell, Richard L. and Alberto L. Sangiovanni-Vincentelli (1987). "Multiple-valued minimization for PLA optimization". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 6.5 (Sept. 1987), pp. 727–50.

Safarpour, Sean, Andreas G. Veneris, Gregg Baeckler, and Richard Yuan (2006). "Efficient SAT-based Boolean matching for FPGA technology mapping". In: *Proceedings of the 43rd Design Automation Conference*. San Francisco, Calif., July 2006, pp. 466–71.

Sapra, Samir, Michael Theobald, and Edmund M. Clarke (2003). "SAT-based algorithms for logic minimization". In: *Proceedings of the 21st IEEE International Conference on Computer Design*. San Jose, Calif., Oct. 2003, pp. 510–17.

Shannon, Claude E. (1949). "The synthesis of two-terminal switching circuits". In: *The Bell System Technical Journal* 28.1 (Jan. 1949), pp. 59–98.

Silva, Luis Guerra e, João P. Marques-Silva, Luis Miguel Silveira, and Karem A. Skallah (1998). "Timing analysis using propositional satisfiability". In: *Proceedings of the 5th IEEE International Conference on Electronics, Circuits and Systems*. Vol. 3. Sept. 1998, pp. 95–98.

Smith, Michael J. S. (1997). *Application-Specific Integrated Circuits*. Boston, Mass.: Addison-Wesley, 1997.

Smith, Alexander, Andreas G. Veneris, Moayad Fahim Ali, and Anastasios Viglas (2005). "Fault diagnosis and logic debugging using Boolean satisfiability". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.10 (2005), pp. 1606–21.

Soeken, Mathias, Alan Mishchenko, Ana Petkovska, Baruch Sterin, Paolo Ienne, Robert K. Brayton, and Giovanni De Micheli (2016a). "Heuristic NPN classification for large functions using AIGs and LEXSAT". In: *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*. Vol. 9710. Lecture Notes in Computer Science. Bordeaux: Springer, July 2016, pp. 212–27.

Soeken, Mathias, Giovanni De Micheli, and Alan Mishchenko (2017). "Busy Man's Synthesis: Combinational delay optimization with SAT". In: *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. Lausanne, Switzerland, Mar. 2017.

Soeken, Mathias, Daniel Große, Arun Chandrasekharan, and Rolf Drechsler (2016b). "BDD minimization for approximate computing". In: *Proceedings of the 21st Asia and South Pacific Design Automation Conference*. Macao, Jan. 2016, pp. 474–79.

Soeken, Mathias, Pascal Raiola, Baruch Sterin, Bernd Becker, Giovanni De Micheli, and Matthias Sauer (2016c). "SAT-based combinational and sequential dependency computation". In: *Proceedings of the 12th International Haifa Verification Conference*. Haifa, Israel, Nov. 2016, pp. 1–17.

Stephan, Paul R., Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli (1996). "Combinational test generation using satisfiability". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 15.9 (1996), pp. 1167–76.

Tang, Kai-Fu, Chi-An Wu, Po-Kai Huang, and Chung-Yang (Ric) Huang (2011). "Interpolation-based incremental ECO synthesis for multi-error logic rectification". In: *Proceedings of the 48th Design Automation Conference*. San Diego, Calif., June 2011, pp. 146–51.

Toda, Takahisa and Takehide Soh (2016). "Implementing efficient all solutions SAT solvers". In: *ACM Journal of Experimental Algorithmics* 21.1 (2016), 12:1–12:44.

Tseitin, Grigorii Samuilovich (1983). "On the complexity of derivation in propositional calculus". In: *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*. Ed. by J. Siekmann and G. Wrightson. Symbolic Computation. Berlin: Springer, 1983, pp. 466–83.

Veitch, Edward W. (1952). "A chart method for simplifying truth functions". In: *Proceedings of the 1952 ACM National Meeting*. Pittsburgh, Pa., 1952, pp. 127–33.

Velev, Miroslav N. (2000). "Formal verification of VLIW microprocessors with speculative execution". In: *Proceedings of the International Conference on Computer Aided Verification*. Chicago, Ill., July 2000, pp. 296–311.

Velev, Miroslav N. and Randal E. Bryant (2003). "Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors". In: *Journal of Symbolic Computation* 35.2 (2003), pp. 73–106.

Venkatesan, Rangharajan, Amit Agarwal, Kaushik Roy, and Anand Raghunathan (2011). "MACACO: Modeling and analysis of circuits for approximate computing". In: *Proceedings of the International Conference on Computer Aided Design*. San Jose, Calif., Nov. 2011, pp. 667–73.

Verma, Ajay K., Philip Brisk, and Paolo Ienne (2009). "Iterative Layering: Optimizing arithmetic circuits by structuring the information flow". In: *Proceedings of the International Conference on Computer Aided Design*. San Jose, Calif., Nov. 2009, pp. 797–804.

Verma, Ajay K., Philip Brisk, and Paolo Ienne (2007). "Progressive Decomposition: A heuristic to structure arithmetic circuits". In: *Proceedings of the 44th Design Automation Conference*. San Diego, Calif., June 2007, pp. 404–9.

Walter, Rouven, Christoph Zengler, and Wolfgang Küchlin (2013). "Applications of MaxSAT in automotive configuration". In: *Proceedings of the 15th International Configuration Workshop*. Vienna, Aug. 2013, pp. 21–28.

Wang, Laung-Terng, Yao-Wen Chang, and Kwang-Ting (Tim) Cheng, eds. (2009). *Electronic Design Automation: Synthesis, Verification, and Test*. San Francisco, Calif.: Morgan Kaufmann Publishers Inc., 2009.

Wegener, Ingo (2000). *Branching Programs and Binary Decision Diagrams: Theory and Applications*. Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics, 2000.

Wood, R. Glenn and Rob A. Rutenbar (1998). "FPGA routing and routability estimation via Boolean satisfiability". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 6.2 (June 1998), pp. 222–31.

Wu, Bo-Han, Chun-Ju Yang, Chung-Yang (Ric) Huang, and Jie-Hong Roland Jiang (2010). "A robust functional ECO engine by SAT proof minimization and interpolation techniques". In: *Proceedings of the International Conference on Computer Aided Design*. San Jose, Calif., Nov. 2010, pp. 729–34.

Yang, Wenlong, Lingli Wang, and Alan Mishchenko (2012). "Lazy man's logic synthesis". In: *Proceedings of the International Conference on Computer Aided Design*. San Jose, Calif., Nov. 2012, pp. 597–604.

Yuan, Jun, Adnan Aziz, Carl Pixley, and Ken Albin (2004). "Simplifying Boolean constraint solving for random simulation-vector generation". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23.3 (Mar. 2004), pp. 412–20.

Curriculum Vitae

Areas of Specialisation

Logic synthesis, Boolean satisfiability, logic visualisation.

Education

2011–2017 **Ph.D. in Computer and Communication Sciences**
École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland
Thesis: Exploiting Satisfiability Solvers for Efficient Logic Synthesis
Supervisor: Paolo Ienne

2007–2011 **B.Sc. in Informatics and Computer Engineering**
Faculty of Electrical Engineering and Information Technologies (FEEIT)
Ss.Cyril and Methodius University, Skopje, Macedonia
Project: Alliance CAD System: Tools and Possibilities

Publications

Conference papers

Zhufei Chu, Xifan Tang, Mathias Soeken, **Ana Petkovska**, Grace Zgheib, Luca Amarù, Yinshui Xia, Paolo Ienne, Giovanni De Micheli, and Pierre-Emmanuel Gaillardon. "Improving circuit mapping performance through MIG-based synthesis for carry chains". In: *Proceedings of the 27th ACM Great Lakes Symposium on VLSI*. Banff, Alberta, Canada, May 2017, pp. 131–36.

Ana Petkovska, Alan Mishchenko, Mathias Soeken, Giovanni De Micheli, Robert Brayton, and Paolo Ienne. "Fast generation of lexicographic satisfiable assignments: Enabling canonicity in SAT-based applications". In: *Proceedings of the International Conference on Computer Aided Design*. Austin, Tex., Nov. 2016, pp. 1–8.

Ana Petkovska, Mathias Soeken, Giovanni De Micheli, Paolo Ienne, and Alan Mishchenko. "Fast hierarchical NPN classification". In: *Proceedings of the 26th International Conference on Field-Programmable Logic and Applications*. Lausanne, Switzerland, Aug. 2016, pp. 1–4.

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". In: *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*. Vol. 9710. Lecture Notes in Computer Science. **Best Paper Award Nominee**. Bordeaux: Springer, July 2016, pp. 212–27.

Ana Petkovska, Grace Zgheib, David Novo, Muhsen Owaida, Alan Mishchenko, and Paolo Ienne. "Improved carry-chain mapping for the VTR flow". In: *Proceedings of the 2015 International Conference on Field Programmable Technology*. Queenstown, New Zealand, Dec. 2015, pp. 80–87.

Ana Petkovska, David Novo, Alan Mishchenko, and Paolo Ienne. "Constrained interpolation for guided logic synthesis". In: *Proceedings of the International Conference on Computer Aided Design*. **Best Paper Award Nominee**. San Jose, Calif., Nov. 2014, pp. 462–69.

BOOK CHAPTERS

Ana Petkovska, Alan Mishchenko, David Novo, Muhsen Owaida, and Paolo Ienne. "Progressive generation of canonical irredundant sums of products using a SAT solver". In: *Advanced Logic Synthesis*. Ed. by Rolf Drechsler and Andre Reis. To appear. Springer, 2017.

REFEREED PAPERS WITHOUT FORMAL PROCEEDINGS

Alan Mishchenko, Robert K. Brayton, **Ana Petkovska**, and Mathias Soeken. "SAT-based optimization with don't-cares revisited". In: *Proceedings of the 26th International Workshop on Logic and Synthesis*. Austin, Tex., June 2017.

Gregoire Hirt, **Ana Petkovska**, and Paolo Ienne. "ELVE: An interactive and extensible visualisation tool for logic circuits". In: *Proceedings of the 4th Workshop on Design Automation for Understanding Hardware Designs*. Lausanne, Switzerland, Mar. 2017.

Ana Petkovska, Alan Mishchenko, Mathias Soeken, Giovanni De Micheli, Robert Brayton, and Paolo Ienne. "Fast generation of lexicographic satisfiable assignments: Enabling canonicity in SAT-based applications". In: *Proceedings of the 25th International Workshop on Logic and Synthesis*. **Best Student Paper Award Nominee**. Austin, Tex., June 2016.

Ana Petkovska, Alan Mishchenko, David Novo, Muhsen Owaida, and Paolo Ienne. "Progressive generation of canonical sums of products using a SAT solver". In: *Proceedings of the 25th International Workshop on Logic and Synthesis*. Austin, Tex., June 2016.

Luca Amarù, **Ana Petkovska**, Pierre-Emmanuel Gaillardon, David Novo Bruna, Paolo Ienne, and Giovanni De Micheli. "Majority-Inverter Graph for FPGA synthesis". In: *Proceedings of the 19th Workshop on Synthesis and System Integration of Mixed Information Technologies*. Jiaosi, Taiwan, Mar. 2015, pp. 165–70.

Ana Petkovska, David Novo, Alan Mishchenko, and Paolo Ienne. "Constrained interpolation for guided logic synthesis". In: *Proceedings of the 23rd International Workshop on Logic and Synthesis*. San Francisco, Calif., May 2014.

Ana Petkovska, David Novo, Ajay K. Verma, Alan Mishchenko, and Paolo Ienne. "Enhancing iterative layering with SAT solvers". In: *Proceedings of the 22nd International Workshop on Logic and Synthesis*. Austin, Tx., June 2013.