Improving the Power of Ordered Binary Decision Diagrams by Integrating Parity Nodes

Dissertation zur Erlangung des akademischen Grades
des Doktors der Naturwissenschaften
am Fachbereich IV der Universität Trier

vorgelegt von Diplom Informatiker
Harald Sack

Oktober 2001

... to Anja and my parents

Our doubts are traitors and make us lose the good we often might win for fearing to attempt.

– Measure for Measure,
William Shakespeare

Acknowledgments

To attract good fortune, spend a new penny on an old friend, share an old pleasure with a new friend and lift up the heart of a true friend by writing his name on the wings of a dragon. *(Chinese Proverb)*

I would like to acknowledge the aid and contribution of *all* the people, whose encouragement and support gave me the motivation and the possibility to undergo the work of this thesis.
First at all I would like to thank my advisor *Prof. Dr. sc. Christoph Meinel*, who gave me the possibility to get things started. He always had an open door for me and many of the ideas developed in this thesis originated from lengthy discussions I had with him. The liberal working atmosphere in his department has given me the necessary freedom to constitute this work at my personal pace. His patience, his humor, and not to forget his good advice (research and otherwise) and his friendship made my time as being his PhD student in Trier very enriching.
I feel a deep sense of gratitude towards *Kerstin*, who always encouraged and supported me in my life, who never complained, when I was focusing my attention on this thesis. I know that neglecting her was a serious mistake. But, after all, I want to thank her for her patience and her friendship.
I would like to express all my gratitude to *Elena Dubrova*, for her support in all subjects related to multiple-valued logic, for her encouragement, and especially for her being such a close friend and for the time we spend together in intriguing and inspiring discussions in Lake Tahoe, Victoria BC., and Trier.
Thanks to my coworkers and friends, esp. *Christian Stangier*, who owns that certain urge of always being a confirmed pessimist that has made him the touchstone in testing the qualities of my ideas - and also in improving my patience, and to *Arno Wagner*, whose obstinacy always encouraged my creativity in finding better ways to explain concepts and ideas. And thanks to both of you for tolerating my sometimes strange sense of humor and sarcasm.
Thanks also to my colleges *Frank Recker*, *Martin Mundhenk*, and *Jochen Bern* for fruitful discussions and proofreading. Also not forgetting to mention *Claudine Trittin* for supporting me in my constant struggle with the English language and her comforting deep friendship. The most difficult task, finally, was to find all bugs in the implementation of the $\oplus$-OBDD package.
I have to thank *Anja* from the bottom of my heart. You are the one, giving me always the support and the inspiration that was mandatory to finish this laborious work.
Also, I would like to express my gratitude towards my parents, who always believed in me and supported every decision of mine. I owe them the seeds to my curiosity, which was always the driving force behind all of my work. Thanks to all my friends for the times you spent allowing me to share my thoughts and ideas and numerous cups of cappuccino.
Besides all personal acknowledgments I would like to thank everyone with whom I had the chance to interact in the course of my education. All of you have enriched my life in so many ways and made it worth while. Please forgive any omissions.

Danksagung

Um das Glück anzuziehen, schenke einem alten Freund einen neuen Pfennig, teile mit einem neuen Freund ein altes Vergnügen, und erhebe das Herz eines wahren Freundes indem Du seinen Namen auf die Flügel eines Drachen schreibst. (Chinesisches Sprichwort)

An dieser Stelle möchte ich mich für die Unterstützung und den Beitrag all derer bedanken, deren Ermutigungen und Beistand mir die Motivation und den Freiraum geschaffen haben, diese Dissertation zu Wege zu bringen.
Als erstes gilt mein Dank meinem Doktorvater *Prof. Dr. sc. Christoph Meinel*, der mir die Möglichkeit bot, zu Anfang als externer Doktorand diese Arbeit anzugehen. Seine Tür stand für mich jederzeit offen und viele der in dieser Dissertation entwickelten Ideen finden ihren Ursprung in unseren mitunter ausgedehnten Diskussionen. Die offene Arbeitsatmosphäre an seinem Lehrstuhl gab mir den nötigen Freiraum, die Arbeit nach eigenem Maßstab und Rythmus zu vollenden. Geduld, Humor, nicht zu vergessen sein guter Rat (nicht nur die Forschung betreffend) und seine Freundschaft waren für mich in meiner Zeit als Doktorand in Trier stets eine Bereicherung.
Mein tiefster Dank gilt vor allem *Kerstin*, die mich in jeder Hinsicht immer unterstützt hat. Ihr verdanke ich auch die Motivation und die Ermutigung, diese Dissertation überhaupt zu beginnen. Mir ist bewußt, daß ich sie über diese Arbeit sehr vernachlässigt habe. Aber, nachdem dieser Abschnitt meines Lebens jetzt beendet ist, danke ich ihr für ihre Geduld und ihre Freundschaft.
Besonders dankbar bin ich auch *Elena Dubrova*, und das nicht nur in allen Belangen rund um die *mehrwertige Logik*. Vielmehr ebenso für ihre Ermutigungen, ihre tiefe Freundsschaft und für die gemeinsame Zeit in Lake Tahoe, Victoria, B.C. und Trier, die mit vielen fesselnden und inspirierenden Diskussionen angefüllt war.
Mein Dank gilt natürlich auch meinen Arbeitskollegen und Freunden, besonders *Christian Stangier*, dessen einzigartige kritisch-pessimistische Sichtweise der Dinge ihn zu einem besonderen Prüfstein hinsichtlich der Qualität meines oft konfusen Ideenuniversums machte - und auch so manches Mal meine Geduld auf die Probe stellte. Danke auch an *Arno Wagner*, dessen hartnäckige Uneinsichtigkeit stets eine Herausforderung für mich war, meine Ideen und Konzepte besser zu durchdenken, zu ordnen und zu erklären. Danke an Euch beide, daß Ihr mir meinen oft so seltsamen Humor und meinen Sarkasmus nie wirklich krumm genommen habt.
Weiterhin möchte ich sowohl meinen Kollegen *Frank Recker*, *Martin Mundhenk* und *Jochen Bern* für so manche fachliche Disskussion wie auch ihr Korrekturlesen. Auf keinen Fall vergessen möchte ich *Claudine Trittin*, die mir stets in allen Widrigkeiten, die die englische Sprache für mich bereit hielt, treu zur Seite stand, sowie für ihre innige Freundschaft.
Am Ende bestand die schwierigste Aufgabe darin, alle Fehler in der Implementierung des $\oplus$-OBDD Pakets aufzuspüren. *Anja*, ich möchte Dir von ganzem Herzen danken, daß Du mir in dieser Zeit immer inspirierend und unterstützend zur Seite standest, um diese mühevolle Aufgabe zu beenden.
Besonders möchte ich mich auch bei meinen Eltern bedanken, die stets jede meiner Entscheidungen in meinem Leben getragen und unterstützt haben. Ich verdanke euch die Wurzeln meiner Neugier, die stets die treibende Kraft hinter all meinem Schaffen war und ist.
Danke auch an all meine hier ungenannten Freunde. Danke für die Zeit, die ich mit euch verbringen durfte, in der ich nicht nur meine Gedanken mit euch teilen durfte, sondern auch so manche Tasse Cappuccino.

Abstract

Hardware bugs can be extremely expensive, financially. Because microprocessors and integrated circuits have become omnipresent in our daily live and also because of their continously growing complexity, research is driven towards methods and tools that are supposed to provide higher reliability of hardware designs and their implementations.
Over the last decade Ordered Binary Decision Diagrams (OBDDs) have been well proven to serve as a data structure for the representation of combinatorial or sequential circuits. Their conciseness and their efficient algorithmic properties are responsible for their huge success in formal verification. But, due to Shannon's counting argument, OBDDs can not always guarantee the concise representation of a given design.
In this thesis, Parity Ordered Binary Decision Diagrams ($\oplus$-OBDDs) are presented, which are a true extension of OBDDs. In addition to the regular branching nodes of an OBDD, functional nodes representing a parity operation ($\oplus$-nodes) are integrated into the data structure, thus resulting in $\oplus$-OBDDs. $\oplus$-OBDDs are more powerful than OBDDs are, but, they are no longer a canonical representation. Besides theoretical aspects of $\oplus$-OBDDs, algorithms for their efficient manipulation are the main focus of this thesis. Furthermore, an analysis on the factors that influence the $\oplus$-OBDD representation size gives way for the development of heuristic algorithms for their minimization. The results of these analyses as well as the efficiency of the data structure are also supported by experiments. Finally, the algorithmic concept of $\oplus$-OBDDs is extended to Mod-p-Decision Diagrams (Mod-p-DDs) for the representation of functions that are defined over an arbitrary finite domain.

Kurzdarstellung

Hardwarefehler können mitunter äußerst kostspielig werden. Da Mikroprozessoren und integrierte Schaltkreise in allen Bereichen unseres täglichen Lebens zunehmend an Bedeutung gewinnen und ihre Komplexität ungebremst wächst, kommt der Forschung nach Methoden und Werkzeugen, die eine höhere Zuverlässigkeit dieser Bausteine gewährleisten, heute eine nicht zu unterschätzende Rolle zu.
Im vergangenen Jahrzehnt konnten sich geordnete binäre Entscheidungsdiagramme (Ordered Binary Decision Diagrams, OBDDs) als state-of-the-art Datenstruktur zur Repräsentation von kombinatorischen und sequentiellen Schaltkreisen durchsetzen und ihre Qualitäten unter Beweis stellen. Kompaktheit und ausgezeichnete algorithmische Handhabbarkeit kennzeichnen einerseits den ungebrochenen Erfolg von OBDDs im Bereich der formalen Verifikation. Auf der anderen Seite aber können OBDDs nicht immer die kompakte Darstellung einer Booleschen Funktionen garantieren.
Gegenstand der vorliegenden Dissertation ist die Analyse von Parity Ordered Binary Decision Diagrams ($\oplus-$OBDDs), eine echte Erweiterung der OBDD Datenstruktur. Zusätzlich zu den herkömmlichen Entscheidungsknoten eines OBDDs werden Funktionalknoten in die Datenstruktur eingeführt, die eine XOR-Operation über das Ergebnis ihrer beiden Vorgängerknoten ausführen und zusammengenommen die Datenstruktur eines $\oplus$-OBDDs definieren. Zwar besitzen $\oplus$-OBDDs mächtigere Darstellungseigenschaften als OBDDs, dieser Vorteil allerdings wird auf Kosten der verlorengegangenen Kanonizität erworben.
Neben der Behandlung theoretischer Aspekte der $\oplus$-OBDD Datenstruktur stehen Algorithmen zur effizienten Manipulation derselben im Mittelpunkt der Arbeit. Darüber hinaus gibt eine Analyse der Faktoren, die für die Abhängigkeit der Darstellungsgröße die Verantwortung tragen, Aufschluss über mögliche heuristische Algorithmen zu deren Minimierung. Sowohl die Ergebnisse dieser Analysen, als auch die Effizienz der vorgestellten Datenstruktur werden experimentell belegt.
Abschließend wird das algorithmische Konzept der $\oplus$-OBDDs auf die Darstellung von Funktionen über endliche Mengen hin zu Mod-p-OBDDs erweitert und Algorithmen zu deren effizienten Manipulation vorgestellt.

Zusammenfassung

Gegenstand der vorliegenden Dissertation ist die Analyse von Parity Ordered Binary Decision Diagrams ($\oplus$-OBDDs), eine Kerndatenstruktur zur Repräsentation digitaler Systeme auf Ebenen mit niedrigem Abstraktionsniveau, die eine echte Erweiterung von Ordered Binary Decision Diagrams (OBDDs, geordnete binäre Entscheidungsdiagramme) darstellen. Die permanent wachsende Nachfrage nach Rechenleistung, die sich in Form erforderlicher Prozessorgeschwindigkeit und Speicherplatzbedarf äußert wird durch die kontinuierlich steigende Komplexität der eingesetzten Softwaresysteme in allen Bereichen unseres täglichen Lebens verursacht und fungiert als treibende Kraft, die die Entwicklung der Hardwareindustrie vorantreibt. Computersysteme sind heute allgegenwärtig: in Transport und Verkehr, im Gesundheitswesen, in Schule und Ausbildung oder in der Industrie - unaufhörlich wächst die Anzahl der Aufgaben, die von Mikroprozessoren und Computern übernommen werden. Integrierte Computersysteme, eingebettet in Flug- oder Kraftfahrzeugen, Mobiltelefonen und selbst in unserem Heim, angefangen mit Unterhaltungselektronik über Küchengeräte bis hin zum *intelligenten* Haus tragen zusammen mit dem überwältigenden Erfolg des Internets maßgeblich dazu bei, diese Systeme heute zu einem unverzichtbaren Bestandteil unseres Lebens zu erklären, wobei sich unsere Abhängigkeit in geradezu bedenklichem Maße tagtäglich vergrößert.

Doch nicht nur die Zahl der Einsatzgebiete von Computersystemen, sondern auch die Komplexität integrierter Schaltkreise als deren Kernkomponenten steigt kontinuierlich.. Der im Herbst 2000 von Intel vorgestellte *Pentium IV Willamette* Mikroprozessor umfasst 42 Millionen Transistoren auf einer Fläche von nur 217 mm^2. *Moore's Gesetz* entsprechend [Moo65] verdoppelt sich die Anzahl der Transistoren auf einem einzelnen Mikrochip alle 18 Monate. Ursprünglich gedacht als bloße Faustregel, basierend auf Gordon Moore's Analysen von 1965, wurde dieses Gesetz zum Leitprinzip der Chipindustrie, die sich dadurch gezwungen sah, immer leistungsfähigere Mikrochips bei gleichzeitigem Preisverfall zu produzieren. Zusätzlich führt der Konkurenzdruck zu immer kurzeren Entwicklungszyklen und fordert von der Industrie immer komplexere Systeme in kürzerer Zeit zu konzipieren. Um einen Hardwaredesigner in die Lage zu versetzen diesem Produktivitätsdruck gerecht zu werden, zentriert sich der Fokus der Entwicklungsarbeit zunehmend auf höheren Abstraktionsebenen. Computergestützte Entwicklungswerkzeuge (Computer Aided Design Tools) gewährleisten dabei die notwendige Automatisierung der unteren Abstraktionsebenen des Designprozesses.

Neben den physikalischen Grenzen, die früher oder später bei anhaltender Miniaturisierung erreicht sein werden, ist es gerade diese explosionsartige Zunahme der Komplexität, die den Beweis der absoluten Korrektheit eines solchen Designs unmöglich macht. Aus diesem Grund geraten auftretende Fehler - zumal bei steigender Abhängigkeit unserer Gesellschaft vom fehlerfreien Funktion-

ieren einzelner Systeme - mehr und mehr in den Blickpunkt der Öffentlichkeit. Die fatalen Konsequenzen eines fehlerhaften Chips innerhalb eines sicherheitskritischen integrierten Systems wie z.B. in Flug- oder Kraftfahrzeugen sind für jedermann offensichtlich. Doch abgesehen davon können auch die finanziellen Auswirkungen, ausgelöst durch fehlerhafte Computersysteme, katastrophale Folgen nach sich ziehen.
Die Bedeutung der Werkzeuge zur computergestützten automatischen Hardware-Verifikation wurde einer breiteren Öffentlichkeit erstmals augenfällig als Ende 1994 der in Intel's Pentium 1 auftretende Floating Point Fehler in der Divisionseinheit des Prozessors (FDIV-Fehler) bekannt wurde und bei Intel, auf Grund der dadurch ausgelösten bislang größten Umtauschaktion in der Geschichte des Personalcomputers, einen Schaden in Höhe von 475 Millionen US-Dollar verursachte [Hof95]. Die Reihe aufgetretener Fehler in Mikroprozessoren reißt nicht ab, auch wenn die Folgen meist weniger spektakulär ausfallen. Zwar existieren zu allen aktuellen Prozessoren Listen mit bekannten, auftretenden Fehlern, doch auch diese mindern häufig die Einsatzfähigkeit des betreffenden Mikrochips nur in einem geringen Maße, da sie nur in selten eintretenden Konfigurationen zum Tragen kommen. Die finanziellen Folgen oder potenziell lebensbedrohlichen Konsequenzen dieser Hardwarefehler ließen sich jedoch vermeiden, wenn sie bereits früh im Designprozess erkannt werden könnten.
Zur Verifikation digitaler Systeme unterscheidet man zwei grundsätzlich verschiedene Ansätze: *empirische Verifikation* und *formale Verifikation*. Der empirische Ansatz verfolgt die Lösung des Problems über die Generierung und Durchführung von Testschemata an einem Modell des entsprechenden Designs. Die Auswirkungen dieser Tests können dann anhand der Ausgaben des Modells analysiert werden. Empirische Methoden sind nicht geeignet, die Korrektheit eines Designs im Sinne einer "Ja/Nein"-Antwort zu belegen. Vielmehr sind sie dazu gedacht, Zuverlässigkeitsaussagen zur Fehlerfreiheit eines Designs entsprechend zu quantifizieren. Auch wenn sich empirische Methoden im frühen Stadium des Fehlersuchprozesses als sehr effektiv erweisen, verlieren sie rasch ihre Effizienz, wenn im Laufe des Prozesses die Fehler immer seltener vorkommen. Eine erschöpfende empirische Simulation eines aktuellen Prozessordesigns ist auf Grund des immensen Zeitbedarfs nicht durchführbar. Daher erlangen formale Methoden zum Nachweis der Korrektheit eines Designs immer stärkere Bedeutung. Formale Verifikation ist dazu bestimmt, absolute Korrektheit eines gegebenen Designs nachzuweisen, bzw. die Zuverlässigkeit einer empirischen Verifikation zu erhöhen, indem Teile des Designs formal verifiziert werden. Ziel ist es dabei, den Nachweis zu führen, dass eine Implementation eines Designs mit einer gegebenen Spezifikation übereinstimmt. Implementation eines Designs bezeichnet hierbei das Modell eines Designs, welches verifiziert werden soll, wohingegen mit der Spezifikation entsprechend abstrakte Modelle bzw. Eigenschaften der Modelle beschrieben werden, mit deren Hilfe die Fehlerfreiheit des Systems nachgewiesen werden soll.
Als Basisgrundlage der Verifikation dient eine entsprechende formale Repräsentation des Modells, um dieses für mathematische Beweisverfahren zugänglich zu machen. Während das Verhalten eines Systems auf einer abstrakten Ebene (*behavioral level*) durch Datenflussdiagramme (*data flow graphs*), Prozess-Algebren

oder höherwertige Logiken erfolgt, umfassen Repräsentationsformen auf niedrigem Abstraktionsniveau (*low level*, *switching level*) endliche Automaten oder Modelle auf Gatter- bzw. Schaltebene. Eine Übersicht der Methoden der formalen Verifikation ist in [Gup92] zusammengestellt.
Der Schwerpunkt dieser Arbeit liegt auf der Analyse von $\oplus$-OBDDs, einer Datenstruktur zur Repräsentation digitaler Systeme auf niedrigem Abstraktionsniveau, die sowohl die symbolische Simulation von kombinatorischen Schaltkreisen auf Gatterebene umfasst, als auch zur Modellierung endlicher Automaten, einschließlich der symbolischen Darstellung riesiger Zustandsräume geeignet ist. Die Datenstruktur, die derzeit bevorzugt für diese Aufgabe Verwendung findet und Bestandteil der meisten kommerziellen Verifikationswerkzeuge ist, sind OBDDs. Allerdings können OBDDs nicht immer effizient zur Darstellung digitaler Systeme eingesetzt werden. Ausschlaggebend dafür ist die Tatsache, dass OBDDs als kanonische Repräsentationsform für Boolesche Funktionen tauglich sind, die u.a. in Form digitaler Schaltkreise vorliegen können. So vorteilhaft sich diese Kanonozität hinsichtlich der algorithmischen Handhabbarkeit erweist, bedingt sie jedoch auch, dass die Darstellung eines Großteils der möglichen Funktionen exponentielle Größe in Relation zur Zahl der Eingabevariablen aufweist [Sha49].
Eine Option, die Darstellungskraft dieser Datenstruktur zu erhöhen, liegt in der Einführung zusätzlicher Funktionalknoten, die eine kompaktere Repräsentation ermöglicht. In der vorliegenden Dissertation werden daher Operatorknoten eingeführt, die eine $\oplus$-Operation (XOR, exclusive or) realisieren und OBDDs hin zu $\oplus$-OBDDs erweitern. Es wird gezeigt, dass $\oplus$-OBDDs tatsächlich sowohl das Potenzial zu einer kompakteren Repräsentation bergen, als auch die algorithmischen Vorteile von OBDDs bewahren. Allerdings sind $\oplus$-OBDDs nicht mehr kanonisch, d.h. eine eindeutige Darstellung einer gegebenen Booleschen Funktion ist nicht mehr gewährleistet, sodass sich der Test der Äquivalenz zweier Boolescher Funktionen über $\oplus$-OBDDs aufwendig gestaltet. Nachfolgend werden effiziente Manipulationsalgorithmen für $\oplus$-OBDDs vorgestellt, die auf einem schnellen probabilistischen Äquivalenztest basieren und deren Leistungsfähigkeit anhand ihres Einsatzes in der symbolischen Simulation von öffentlich erhältlichen Standard-Benchmarkschaltkreisen nachgewiesen wird. Darauf aufbauend werden Methoden zur weiteren Effizienzsteigerung erörtert, die sowohl ein dynamisches Umordnen der Variablen, als auch die gezielte Plazierung von $\oplus$-Knoten innerhalb der $\oplus$-OBDD Datenstruktur umfassen. Zuletzt wird die Datenstruktur dahingehend erweitert, auch Funktionen, die über endlichen Mengen definiert sind, repräsentieren zu können.
Die vorliegende Untersuchung gliedert sich wie folgt: Nach einer ersten Motivation und Hinführung zum Thema werden in Kapitel 2 grundlegende Definitionen und Eigenschaften von OBDDs wiederholt. Neben komplexitätstheoretischen Eigenschaften elementarer OBDD-Manipulationsalgorithmen liegt der Schwerpunkt dieses Kapitels einerseits auf implementationstechnischen Details, andererseits auf grundlegenden Optimierungstechniken.
Im nächsten Kapitel wird in einem ersten Schritt die Notwendigkeit der Erweiterung der OBDD-Datenstruktur motiviert. Das Hauptaugenmerk richtet sich bei derartigen Verallgemeinerungen auf die Kapazitätserweiterung hin-

sichtlich der Darstellungskraft der Datenstruktur. In der Regel liegt allerdings die Schwäche der meisten Erweiterungen in ihrer mangelhaften algorithmischen Handhabbarkeit. Darüberhinaus sind diese Verallgemeinerungen auch meist nicht mehr kanonisch, d.h. ein Äquivalenztest für diese Datenstrukturen gestaltet sich meist sehr aufwendig. Als Schwerpunkt wird die Einführung zusätzlicher Knoten von unterschiedlicher Funktionalität betrachtet, die neben der Lockerung von Restriktionen, wie z.B. die Aufrechterhaltung einer fixen Variablenordnung auf allen Berechnungspfaden oder der Beschränkung, dass jede Variable auf einem beliebigen Berechnungspfad höchstens einmal getestet werden darf, eine Generalisierung der OBDD-Datenstruktur bedeutet. Es wird gezeigt, dass nur die Operatoren $\oplus$ (XOR) und $\equiv$ (EQUIVALENCE) in der Lage sind die erforderlichen algorithmischen Vorteile von OBDDs zu bewahren. Daher werden in Kapitel 4 $\oplus$-OBDDs eingeführt, basierend auf OBDDs mit zusätzlichen $\oplus$-Funktionalknoten. Zunächst werden die Darstellungseigenschaften von $\oplus$-OBDDs analysiert, die ebenfalls komplexitätstheoretische Ergebnisse beinhaltet, um einen Vergleich mit konkurrierenden Datenstrukturen wie OBDDs, OFDDs, FBDDs oder ESOPs ermöglichen zu können. Darüber hinaus werden Zusammenhänge zwischen diesen Repräsentationsformen und $\oplus$-OBDDs geknüpft, was notwendigerweise eine effiziente Simulation derselben durch $\oplus$-OBDDs umfasst. Im anschließenden Abschnitt wird ein schneller probabilistischer Äquivalenztest für $\oplus$-OBDDs vorgestellt und gezeigt, wie dieser die effiziente Umsetzung von $\oplus$-OBDD Manipulationsalgorithmen gewährleistet. Basierend auf diesen Manipulationsalgorithmen wird die erfolgreiche Implementation eines Programmpakets zur Manipulation von $\oplus$-OBDDs beschrieben, dessen Effizienz anhand der symbolischen Simulation von kombinatorischen und sequentiellen Schaltkreisen des Benchmarkpakets nachgewiesen und mit den Ergebnissen eines Standard-OBDD-Manipulationspakets verglichen.
Das nachfolgende Kapitel dreht sich um die Möglichkeiten einer weiteren Optimierung der $\oplus$-OBDD Datenstruktur. Neben der gewählten Variablenordnung, die den entscheidenden Faktor hinsichtlich des Speicherplatzbedarfs für OBDDs darstellt, ist es für $\oplus$-OBDDs ebenfalls von entscheidender Bedeutung, wieviele zusätzliche $\oplus$-Operatorknoten eingeführt werden und an welchen Stellen innerhalb des $\oplus$-OBDDs diese positioniert werden. Im Folgenden werden Notwendigkeiten und Beschränkungen analysiert, die zur dynamischen Adaption der Variablenordnung von $\oplus$-OBDDs beachtet werden müssen. Desweiteren werden implementationstechnische Aspekte zur Realisierung eines $\oplus$-OBDD Pakets, einschließlich zugehöriger Optimierungsmethoden diskutiert, die die Grundlage einer effizienten Realisierung bilden. Ausgehend von den gewonnenen Erkenntnissen werden heuristische Verfahren zur Minimierung von $\oplus$-OBDDs entwickelt und im Anschluss experimentell erprobt.
Kapitel 6 beleuchet einen weiteren und abschließenden Aspekt in der Verallgemeinerung der Datenstruktur. Hier werden $\oplus$-OBDDs, ausgehend als Repräsentationsform für Boolesche Funktionen hinsichtlich des zugrunde liegenden Definitionsbereichs zu Funktionen über einem endlichen Körper hin erweitert. Die daraus resultierende Datenstruktur, Mod-p-Decision Diagrams (Mod-p-DDs), stellt ebenfalls eine Verallgemeinerung von Multi Valued Decision Diagrams (MDDs) dar, indem in die MDD-Datenstruktur zusätzliche Funktional-

knoten integriert werden. Diese mod-p Funktionalknoten repräsentieren ihrerseits eine Addition modulo p und können somit als Generalisierung der binären $\oplus$-Operatorknoten in $\oplus$-OBDDs betrachtet werden. Mod-p-DDs sind genau wie $\oplus$-OBDDs nicht mehr kanonisch. Der probabilistische Äquivalenztest, der für $\oplus$-OBDDs in Kapitel 4 dargelegt wurde, wird entsprechend für Mod-p-DDs erweitert und angepasst. In einem letzten Schritt werden Manipulationsalgorithmen für Mod-p-DDs eingeführt und Hinweise zu einer möglichen effizienten Implementierung gegeben.
Das abschließende Kapitel resümiert noch einmal die Hauptergebnisse der Dissertation und skizziert einen Ausblick auf weiterführende Arbeiten.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis presents $\oplus$-Ordered Binary Decision Diagrams ($\oplus$-OBDDs), a functional extension of Ordered Binary Decision Diagrams, which serve as a core data structure for the switching level representation of digital systems. The main objective is the analysis of $\oplus$-OBDDs and their properties, developing an efficient implementation, and the evaluation of their performance. In this chapter we will demonstrate the need for efficient verification tools as an aid for the design of digital systems and give an overview of the key points of the thesis.

1.1 Motivation

Today, a steadily growing demand on computing power that is manifesting itself in terms of mere microprocessor performance needs and storage requirements, driven by a simultaneous growth in the complexity of software systems in every area of our today's life, constitutes the driving force of the so called silicon industry. The ubiquitous invasion of hardware systems in traffic and transportation, health care and surgery, education and manufacturing - everywhere, the number of tasks taken over by microprocessors and computers is subject to a permanent increase. Embedded systems inside automobiles, airplanes, cell phones, even in our homes, starting from consumer electronics to kitchen aids and environmental facilities, not to mention the striking success of the Internet have let these systems become critical parts of our daily lives now, and we will become increasingly dependent on them.
Not only that the number of areas, where computers are utilized is permanently growing, but also the integrated circuits as their basis multiply in their inherent complexity. In fall 2000 the Intel Corporation presented the Pentium IV Willamette, which combines 42 million transistors on an area of 217 mm^2. According to *Moore's Law* [Moo65] the number of transistors on a single chip doubles every eighteen months, and it has withstood the test of time since Gordon Moore made this observation in 1965 [Moo95]. While originally intended as a rule of thumb, Moore's Law has become the guiding principle for the industry to deliver ever-more-powerful semiconductor chips at proportionate decreases in cost.

In the midst of the growing complexity of these designs, there is also a tremendous pressure to get early time-to-market schedules to maintain business competitiveness. This leads to the need of tools that enable the companies to design even more complicated systems in lesser time. In order to provide the hardware designer with a higher level of productivity, the focus of the design effort has moved towards higher levels of abstractions, which was only possible by the introduction of Computer Aided Design (CAD) tools that automate the design process at lower levels of abstraction.

Besides physical limitations that will be reached sooner or later, this explosion in complexity is responsible for the fact that guaranteeing absolute correctness of hardware designs has become an infeasible task. But, as we are depending more and more on the operational correctness of the computer systems that are surrounding us, discovered faults and flaws in processor designs are coming to the public's attention. The disastrous effects of adding a faulty chip to an embedded system inside an airplane or an automobile are obvious. Apart from these safety critical applications the financial costs caused by faults in computers might get prohibitively expensive. The importance of hardware verification tools became obvious, when in late 1994 the bug in Intels Pentium I floating point division unit (FDIV-bug) was uncovered, which was responsible for a $475 million loss for the Intel Cooperation because it started the largest recall action in the history of computing [Hof95]. Actually, only five missing transistors out of 3.3 Million transistors caused floating point errors in 42% of all personal computers sold in December 1994.

Then, by the time when I started writing this thesis, the latest important hardware error, which was the Intel memory translator hub (MTH) error in May 2000 should serve as an actual motivation for this work. Again, the situation was not only embarrassing, but also rather costly for the Intel Cooperation, because it necessitates a recall of the motherboards using the error prone MTH. But, while writing the introductory chapter one year later, Intel was struck by another disaster concerning their most expensive 32-Bit microprocessor, the Pentium III Xeon 900MHz with 2MB level 2 cache, where a rarely occurring error might cause the processor to step into an endless loop. Not only the market leader, but, also other companies are struck by errors in their produced hardware, e.g. at the same time, Sun Microsystems and its UltraSPARC III microprocessor, where firmware patches have become necessary to fix an error in the floating point unit that caused stale data during prefetch operations.

Taking into account the general growth in complex computer systems and the risk that is connected to errors in these systems, risks that are not only connected to immense costs, but also to possible life threatening consequences, the importance to guarantee the correctness of an implemented hardware design can not be underestimated. An early detection of these bugs would have saved Intel Cooperation from this huge financial loss and also from the simultaneous public relations fiasco.

1.2 Verification of Digital Systems

The existing methods for the verification of digital systems can be divided into two categories: empirical and formal methods. The empirical approach attacks the problem of design verification by generating and applying tests to a model of the design. Then, the effect of the input tests is simulated with the model. Empirical methods are not attempting to prove correctness of designs in terms of a yes/no answer, but rather to derive a level of confidence that the design is free of errors. Although empirical methods are provably effective in the very early stages of the debugging process, when a system is infected with multiple bugs, their effectiveness quickly decreases when the design becomes less erroneous. Exhaustive simulation of today's microprocessor designs would require an unsurmountable amount of time. Thus, the task of formal verification has become the main focus of attention. Formal verification is giving way to proving either absolute correctness of a given design or, at least, is able to increase the reliability of empirical simulation by formally verifying parts of the entire design. Methods of formal verification are aiming at establishing that an implementation satisfies a given specification. While implementation refers to a model of the design to be verified, the term specification refers to a more abstract model or some properties with respect to which the correctness is to be determined. A formal model of the underlying design with a precisely defined meaning enables the application of mathematical proof methods. While on a behavioral level this formalism is achieved in terms of data flow graphs, process algebras, and higher order logics, lower level formalism comprises finite state machines and switching level models. The design can be directly modeled in one of these formalisms, which form the basis of formal verification methods. For a survey of formal verification methods see [Gup92].

1.3 Scope of the Thesis

The focus of this thesis lies in the analysis of $\oplus$-OBDDs, a data structure for the representation of models of digital systems in the lower levels of abstraction. On this level of abstraction $\oplus$-OBDDs can be utilized for the representation of combinatorial circuits on the gate level (*symbolic simulation*) or also for the symbolic representation of finite state machines including large state spaces. Today, most CAD tools are employing OBDDs for these tasks, because OBDDs have well proven their efficiency over the last decade. But, not in every case the efficient application of OBDDs can be guaranteed. As being a canonical data structure for the representation of Boolean functions, they suffer from the potential of an exponential blow up in size and then, prevent a proper verification of a given design. We extend the concept of OBDDs to $\oplus$-OBDDs by the insertion of operator nodes representing the Boolean parity function ($\oplus$, XOR, EXOR). $\oplus$-OBDDs have the potential of being more efficient, while simultaneously preserving the excellent algorithmic properties of OBDDs. $\oplus$-OBDDs are not canonical. So, there does not necessarily exist a unique $\oplus$-OBDD representation for a given Boolean function and thus, testing the functional equivalence

of two designs given in terms of $\oplus$-OBDDs becomes a complex task. We present manipulation algorithms for $\oplus$-OBDDs based on a fast probabilistic equivalence test and prove their efficiency in symbolic simulation of publicly available standard benchmarks. Furthermore, we show how to improve $\oplus$-OBDD efficiency by adapting variable reordering algorithms and by proper $\oplus$-node placement. Finally, we generalize the $\oplus$-OBDD concept from the Boolean domain to an arbitrary finite domain by introducing Mod-p-Decision Diagrams (Mod-p-DDs) and show how to adapt manipulation algorithms efficiently for this new data structure.

1.4 Overview of the Thesis

We start our analysis in Chapter 2 with a recapitulation of preliminary basic definitions of OBDDs and their related manipulation algorithms. Besides complexity results for elementary operations on OBDDs, we focus on important details about their implementation and give a brief overview on available optimization techniques.
In Chapter 3 we motivate the need for extensions of the OBDD data structure. The main objective of all generalizations is to make the data structure more efficient in terms of their capacity in representation. Unfortunately, most extensions suffer from the disadvantage of a more complex algorithmic behavior concerning their manipulation. Another disadvantage is that generalizations often lead to non canonicity and thus, making equivalence testing often rather difficult. Besides losing restrictions as reading each variable at most once, or keeping the same fixed variable order on every computation path, we concentrate on the possibility of introducing additional operator nodes of distinct functionality. We show that only the operators $\oplus$ and $\equiv$ are able to preserve the required algorithmic properties.
Thus, in Chapter 4 we introduce $\oplus$-OBDDs as OBDDs with additional $\oplus$-operator nodes. First, we give an analysis of $\oplus$-OBDD properties and give some complexity related results for a comparison with other competitive data structures as there are OBDDs, OFDDs, FBDDs, or ESOPs. We also show how these data structures are connected to $\oplus$-OBDDs and how $\oplus$-OBDDs can be transformed into these representations. In the next step, we show, how to implement a fast probabilistic equivalence test for $\oplus$-OBDDs and how to apply this test in $\oplus$-OBDD manipulation algorithms that are introduced in the following section. Based on these algorithms we present the implementation of a $\oplus$-OBDD package and show its efficiency in symbolic simulation compared to standard OBDDs.
Chapter 5 is focussed on possibilities of $\oplus$-OBDD optimization. Besides the variable order that is the most influencing factor for OBDD size, in $\oplus$-OBDDs it is also necessary to introduce an appropriate number of $\oplus$-nodes at well chosen and distinct positions. We show, how to adapt variable reordering algorithms to the requirements and limitations of $\oplus$-OBDDs and analyze the influence of proper $\oplus$-node introduction and placement.
As a next step of generalization in Chapter 6, we extend $\oplus$-OBDDs from the

Boolean domain to an arbitrary finite domain and introduce so called Mod-p-OBDDs, which correspond to Multiple Valued Decision Diagrams with additional operator nodes that are representing addition modulo p. We show how to extend the probabilistic equivalence test from $\oplus$-OBDDs to Mod-p-DDs and give the key manipulation algorithms for Mod-p-DDs.
Chapter 7 concludes the thesis with summarizing its key results and gives hints on possible future work on the subject.

1.5 Publications

Parts of this thesis are contained in the following publications:

1. H. Sack, Ch. Meinel: Improving XOR-Node Placement for Mod2OBDD Minimization, *Proc. of the 5th Int. Workshop on Applications of the Reed-Muller Expansion in Circuit Design* (Reed-Muller 2001), Mississippi State University, Starkville, Mississippi, USA, 2001.

2. H. Sack, Ch. Meinel: A Simple Heuristic for Mod2OBDD Minimization, *in Proc. of IEEE/ACM Int. Workshop of Logic And Synthesis* (IWLS2001), Lake Tahoe, CA, USA, 2001, pp.304-309.

3. H. Sack, E. Dubrova, Ch. Meinel: Representation of Multiple-Valued Functions with Mod-p-Decsion Diagrams, *in Proc. of IEEE/ACM Int. Workshop of Logic Synthesis* (IWLS2000), Dana Point, CA, USA, 2000, pp. 341-348.

4. H. Sack, E. Dubrova, Ch. Meinel: Mod-p Decision Diagrams: A Data-Structure for Multiple-Valued Functions, *in Proc. of the 30th IEEE International Symposium on Multiple-Valued-Logic* (ISMVL 2000), Portland, Oregon, USA, 2000, pp. 233-238.

5. E. Dubrova, H. Sack: Probabilistic Verification of Multiple-Valued Functions, *in Proc. of the 30th IEEE International Symposium on Multiple-Valued-Logic* (ISMVL 2000), Portland, Oregon, USA, 2000, pp. 460-466.

6. Ch. Meinel, H. Sack: Mod2OBDDs - a BDD Structure for Probabilistic Verification, *in Electronic Notes in Theoretical Computer Science*, vol.**22**, 2000.

7. Ch. Meinel, H. Sack: Algorithmic Considerations for Parity-OBDD Reordering, *in Proc. of the 1999 IEEE/ACM Int. Workshop on Logic Synthesis* (IWLS99), Lake Tahoe, CA, 1999, pp.71-74.

8. Ch. Meinel, H. Sack: Case Study: Manipulating Mod2OBDDs by Means of Signatures, *Proc. of the 3rd Int. Workshop on Applications of the Reed-Muller Expansion in Circuit Design* (Reed-Muller'97), Oxford, UK, 1997, pp. 175-184

Chapter 2

Preliminaries

In this chapter we start with an introduction into the theoretical context of this work and motivate, why it is reasonable to work with Ordered Binary Decision Diagrams and their functional extensions. In particular, we survey some important state-of-the-art techniques in the area of Boolean manipulation that form the foundation for our analysis and the starting point for the development of a new data structure.

2.1 Ordered Binary Decision Diagrams

All tasks involved in the process of computer aided design of very large scale integrated circuits (VLSI-CAD) and Electronic Design Automation (EDA) are based on the manipulation of *switching functions*. Switching functions are a subset of Boolean functions, restricted to the domain $\{0,1\}$ and are defined in the following way:

Definition 2.1 *A Boolean function* $f : \{0,1\}^n \to \{0,1\}^m$, $n, m \in \mathbb{N}$ *over a set of* n *input variables* $\{x_1, \ldots, x_n\}$ *is called* **switching function**.

For working efficiently with switching functions, we require some sort of data structure for the representation of these functions within a computer. Basically, this data structure should describe the function under consideration appropriately and completely. Besides this basic requirement that has to be fulfilled in every case the data structure should be concise for fitting into the computer's memory, and in addition manipulation and evaluation of the represented function should be simple.
The general problem that is connected to the representation of switching function lies in the immense number of possible functions of n variables (also denoted as inputs) and m outputs, which computes to $2^{m \cdot 2^n}$. Therefore, a data structure that determines all possible switching functions uniquely has to be of exponential size for a most of the functions to be represented.
Boolean functions are finite functions. Therefore, in principle it is possible to represent Boolean functions with the help of *truth tables*, i.e. tables where all possible variable assignments together with the computed function values are listed. But, because of the fact that a truth table for a given function

$f : \{0,1\}^n \rightarrow \{0,1\}$ of $n \in \mathbb{N}$ input variables requires 2^n rows, truth tables are only suitable for functions with a rather small number of variables. On the other hand, truth tables are easy to manipulate from an algorithmic point of view. The application of an arbitrary binary Boolean operator $\otimes$ to two functions f, g, $f \otimes g$, both given in terms of truth tables requires only time linear in the number of input rows. But, the number of input lines is always exponential in the number of input variables and therefore, the requirement of being concise can never be fulfilled for achieving any relevance in practice.
Boolean formulas, i.e. literals connected by Boolean operators are another way for representing Boolean functions. Compared to truth tables they are much more concise, but performing manipulation operations like testing, whether there exists a variable assignment $a \in \{0,1\}^n$ for $f : \{0,1\}^n \rightarrow \{0,1\}$ that computes $f(a_1, \ldots, a_n) = 1$ (*satisfiability test*) or testing, whether two Boolean functions f, g, given in terms of Boolean formulas are representing the same function (*equivalence test*) are NP-complete.
Therefore, the requirements for a well suited data structure for the computer internal representation of Boolean functions are quiet clear: First, the data structure should be concise for as many Boolean functions of practical relevance as possible and secondly, manipulation operations should be efficiently feasible from an algorithmic point of view.
Instead of representing Boolean functions based on computation rules as in the models mentioned above, *branching programs* are a data structure that is using a decision process for the representation of Boolean functions.

Definition 2.2 *A* **branching program** *is a rooted directed acyclic graph $G = (V, E)$ that is representing a Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$. $G = (V, E)$ consists out of two sets of nodes $V = V_B \cup V_T$, $V_B \cap V_T = \emptyset$, and a set of edges $E \subseteq V \times V$. The branching nodes V_B are labeled with a Boolean variable $x_i \in \{x_1, \ldots, x_n\}$ and are switching according to a given variable assignment $a_i \in \{0,1\}$ to one of their two successor nodes. The terminal nodes V_T are labeled with a Boolean constant 0 or 1. They have no outgoing edges and denote the function value $f(a_1, \ldots, a_n) = b \in \mathbb{N}$ that is computed by the branching program for a given variable assignment $(a_1, \ldots, a_n) \in \{0,1\}^n$.*

The introduction of branching programs for the representation of switching circuits dates back to Lee in the late 1950's [Lee59] and was later refined as a representation for Boolean functions by Akers resulting in the Binary Decision Diagram concept [Ake78]. By postulating several restrictions on branching programs they turn into a data structure that fulfills the above mentioned properties quite well for the representation of switching functions within computers - and we finally end up in so called Ordered Binary Decision Diagrams (OBDDs), today's state-of-the-art data structure for the representation of Boolean functions in VLSI CAD.

2.1.1 Definitions and Properties

Definition 2.3 *Let $X = \{x_0, \ldots, x_{n-1}\}$ be a set of n Boolean variables and let $\pi : X \rightarrow \{0, 1, 2, \ldots, |X| = n - 1\}$ be a bijective mapping of the variable*

Figure 2.1: OBDDs for $f = \overline{x}y + x(y\overline{z} + \overline{y}z)$

indices. An **Ordered Binary Decision Diagram (OBDD)** *[Bry86, Bry92] is a rooted directed acyclic graph with the following properties: There are two distinct terminal nodes (*sink nodes*) labeled with the Boolean constants 0 and 1. All non terminal nodes (*branching nodes, inner nodes*) are labeled with a Boolean variable x_i, $i \in \{0, \ldots, n-1\}$ and have two outgoing edges labeled with 0 (*0-edge*) and 1 (*1-edge*), respectively. The order in which the variables occur in the diagram is consistent with the variable order given by π, i.e. if there is an edge leading from a node labeled by x_i to a node labeled by x_j, then $\pi(x_i) < \pi(x_j)$ must hold. On all paths from the root to a sink in the OBDD all variables must occur at most once (*read-once property*).*

For computing the function value of a function f given in terms of a branching program for a given variable assignment $(a_0, \ldots, a_{n-1})$, $a_i \in \{0, 1\}$ one is following a path starting in the root, switching at each node to the edge given by the according variable assignment $x_i = a_i$. The label of the reached sink determines the value of the function on that specific input. Thus, the evaluation of a function value of a Boolean function given in terms of an OBDD for a given assignment requires time $O(n)$. Figure 2.1 shows two OBDDs for the function $f : \{0,1\}^3 \to \{0,1\}$, $f = \overline{x}y + x(y\overline{z} + \overline{y}z)$. Note that in the diagrams 1-edges are denoted with a solid line and 0-edges with a dotted line. As shown in Fig. 2.1 with our definition of OBDDs it is possible to construct several different OBDDs representing the same Boolean function. But, for working efficiently with OBDDs a unique representation for Boolean functions is required.

An OBDD is called *reduced* if it does not contain any vertex v such that the 1-edge and the 0-edge are pointing to the same node, and it does not contain any distinct vertices v and v' such that the subgraphs rooted by v and v' are isomorphic. Equivalently, an OBDD is called reduced if none of the following two *local reduction rules* can be applied:

Deletion rule: (*simple reduction*) If the 1-edge and the 0-edge of a node v lead to the same node w, then eliminate v and redirect all incoming edges to w.

Merging rule: (*algebraic reduction*) If the nodes v and v' are labeled with the same variable, their 1-edge lead to the same node w_1, and their 0-edge

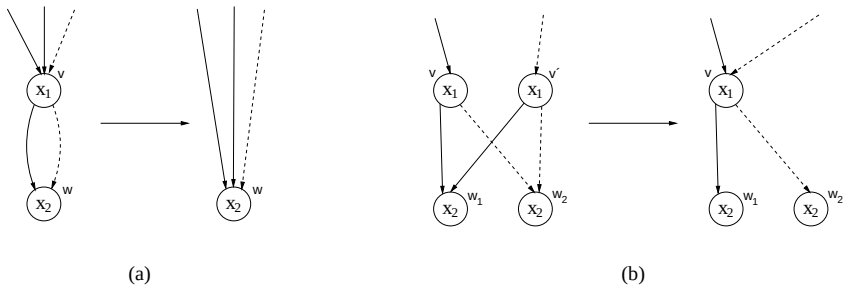


Figure 2.2: Deletion Rule (a) and Merging Rule (b) for OBDDs.

lead to the same node w_2, eliminate one of the two nodes v, v' and redirect all incoming edges to the other node.

The ongoing application of the local reduction rules to an OBDD until no further reduction can be applied is well suited for an algorithmic implementation. Both reduction rules are shown in Figure 2.2.
Sieling and Wegener have shown that given an arbitrary not reduced OBDD G, it can be transformed into a completely reduced OBDD G' in time $O(|G|)$ [SW93]. By the size $|G|$ of an OBDD G we denote the number of its inner nodes.
Reduced OBDDs fulfill the fundamental property of being a canonical representation for Boolean functions.

Fact 2.1 *With respect to a given variable order π the reduced OBDD P_f for each Boolean function $f : \{0,1\}^n \to \{0,1\}$ is uniquely determined (up to isomorphism) and can be computed from an arbitrary OBDD G_f for f in time $O(|G_f|)$.*

For the representation of multiple output switching functions $f : \{0,1\}^n \to \{0,1\}^m$, $m > 1$ we can use multi-rooted OBDDs called *shared OBDDs*, where each root G_{f_i}, $1 \leq i \leq m$ is representing a subfunction $f_i : \{0,1\}^n \to \{0,1\}$ of f (see Figure 2.3). In the following all functions are represented by shared OBDDs and we will always refer to reduced OBDDs w.r.t. a given variable order π, while only using the term OBDD.
In each node v of an OBDD G labeled with x_i a *Boole/Shannon-decomposition* (also simply referred to as *Shannon-Expansion*) w.r.t. the variable x_i is computed. If the node v is representing the Boolean function $f(x_1, \ldots, x_n)$, then, the OBDDs rooted by the two successors v_0, v_1 of v are representing

$$f_{v_0} = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$$

and

$$f_{v_1} = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n),$$

respectively.

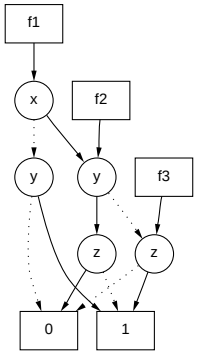


Figure 2.3: Shared OBDD.

Definition 2.4 *The subfunction (restriction) $f|_{x_i=a}$, $a \in \{0,1\}$ that is defined by the replacement of a variable x_i by a Boolean constant a, is defined by*

$$f|_{x_i=a}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, a, x_{i+1}, \ldots, x_n)$$

$f|_{x_i=1} = f|_{x_i}$ is called the *positive cofactor* of f w.r.t. x_i, and $f|_{x_i=0} = f|_{\overline{x}_i}$ is called the *negative cofactor*, respectively. The operation *replacement of an arbitrary variable x_i by a Boolean constant a* in an OBDD G can be computed in time $O(|G|)$. Now, the function f can be written with Boole/Shannon-decomposition

$$f = x_i f|_{x_i} + \overline{x_i} f|_{\overline{x_i}}.$$

OBDDs are not only a canonical representation of Boolean functions, but they can also be manipulated rather efficiently. For basic Boolean manipulation tasks that have to be performed very frequently in CAD systems as e.g. testing satisfiability, testing equivalence, composition (i.e. substitution of a variable by another Boolean function), or Boolean synthesis for Boolean functions given in terms of OBDDs there exist algorithms with runtime polynomial in the number of input variables [Bry86, BRB90].
The most important basic operations are:

1. **Evaluation**:
   Given an OBDD representation G of $f : \{0,1\}^n \to \{0,1\}$ and an assignment $a \in \{0,1\}^n$ compute $f(a)$.

2. **Satisfiability**:
   Given an OBDD representation G of $f : \{0,1\}^n \to \{0,1\}$, is there an assignment $a \in \{0,1\}^n$ such that $f(a) = 1$?

3. **Replacement by constants**:
   Given an OBDD representation G of $f : \{0,1\}^n \to \{0,1\}$, a variable $x_i \in \{x_1, \ldots, x_n\}$, and a constant $a \in \{0,1\}$, compute $f|_{x_i=a}$.

4. **Equivalence**:
   Given OBDD representations G_f, G_g of $f, g : \{0,1\}^n \to \{0,1\}$, decide whether $f = g$ holds.

5. **Binary synthesis**:
   Given the OBDDs G_f and G_g representing $f, g : \{0,1\}^n \to \{0,1\}$ and a Boolean operation $\otimes : \{0,1\} \times \{0,1\} \to \{0,1\}$, compute an OBDD representation G_h of $h = f \otimes g$.[1]

6. **Complementation**:
   Given an OBDD representation G of $f : \{0,1\}^n \to \{0,1\}$, compute an OBDD representation of $\overline{f}$.

7. **Universal Quantification**:
   Given an OBDD representation G of $f : \{0,1\}^n \to \{0,1\}$ and a variable $x_i \in \{x_1, \ldots, x_n\}$, compute the OBDD representation of $\forall x_i : f$.

8. **Existential Quantification**:
   Given an OBDD representation G of $f : \{0,1\}^n \to \{0,1\}$ and a variable $x_i \in \{x_1, \ldots, x_n\}$, compute the OBDD representation of $\exists x_i : f$.

9. **Composition**:
   Given the OBDDs G_1, G_2 of $g_1, g_2 : \{0,1\}^n \to \{0,1\}$ and an arbitrarily chosen variable $x_i \in \{x_1, \ldots, x_n\}$, compute the OBDD representation H of $h = g_1|_{x_i = g_2}$.

Some of these operations are not independent of each other, e.g. satisfiability of the function f can also be tested as the negation of testing the equivalence of f and the constant 0, while equivalence of two functions f and g can be performed by XOR-synthesis of f and g, followed by the negation of testing the satisfiability of the result. For OBDDs the time complexity of all basic manipulation tasks is polynomially related to the size of the input OBDDs.

Fact 2.2 *Let G, G_1, and G_2 be OBDDs ordered w.r.t. a variable ordering π that are representing the Boolean functions $g, g_1, g_2 : \{0,1\}^n \to \{0,1\}$, and let $x_i \in \{x_1, \ldots, x_n\}$ an arbitrary variable.*

(1) The evaluation of G can be computed in time $O(n)$, where n is the number of variables.

(2) Satisfiability of G can be tested in time $O(1)$.

(3) Replacement of an arbitrary variable x_i in G by a constant can be carried out in time $O(|G|)$. The resulting graph G' is again ordered by π and it holds that $|G'| \leq |G|$.

(4) Equivalence of G_1 and G_2 can be decided in time $O(|G_1| + |G_2|)$. Note that if G_1 and G_2 are represented by a common shared OBDD the equivalence test is possible in constant time.

Let $\otimes$ denote an arbitrary Boolean operation $\otimes : \{0,1\} \times \{0,1\} \to \{0,1\}$, e.g. disjunction or conjunction. In order to compute $f \otimes g$ from given OBDDs of the

[1] In a similar way the synthesis of three or more OBDDs can be defined.

Boolean functions f and g, we can also apply the Boole/Shannon-decomposition w.r.t. the first variable x of the given variable order π:

$$f \otimes g = x(f_x \otimes g_x) + \overline{x}(f_{\overline{x}} \otimes g_{\overline{x}})$$

$f \otimes g$ can be computed by repeated application of this decomposition. In order to perform this task efficiently, multiple calls with the same pair of arguments are avoided - instead, the previously computed results will be looked up in a table. By using this technique, the originally exponential number of decompositions is now bounded by the product of the sizes of both OBDDs, and thus, is polynomial.

Fact 2.3

(5) Boolean synthesis of G_1 and G_2 with an arbitrary Boolean operation can be carried out in time $O(|G_1| \cdot |G_2|)$.

(6) Complementation of G can be computed in constant time $O(1)$.

(7) Universal Quantification of G w.r.t. an arbitrary variable x_i can be carried out in time $O(|G^2|)$. Note that universal quantification is defined by $\forall x_i f = f_{x_i} \cdot f_{\overline{x_i}}$.

(8) Existential Quantification of G w.r.t. an arbitrary variable x_i can be carried out in time $O(|G^2|)$. Note that existential quantification is defined by $\exists x_i f = f_{x_i} + f_{\overline{x_i}}$.

(9) Composition of G_1 by replacement of an arbitrary variable x_i by G_2 can be computed in time $O(|G_1|^2 \cdot |G_2|)$.

2.1.2 Implementation Techniques

For working with OBDDs in a practical environment an efficient implementation of the data structure itself as well as sophisticated algorithms for all involved manipulation operations are of essential importance. We have already shown that synthesis of two OBDDs, if implemented in a straightforward way requires run time exponential in the number of input variables. But, for all manipulation tasks polynomial run times are necessary and thus, we have to think of more sophisticated ways of implementation.

Hash Tables and Canonicity

Because OBDDs are a canonical representation of Boolean functions, we have to assure that equivalent OBDDs can easily be determined. Two OBDDs P_1 and P_2 are equivalent, if their root nodes v_1 and v_2 are labeled with the same variable x and if the successors of both root nodes $P_{1_x}, P_{1_{\overline{x}}}$ and $P_{2_x}, P_{2_{\overline{x}}}$ are equivalent. Thus, we can identify two OBDDs by comparing the variable in the root node and their two successors. If two OBDDs are equivalent, their root nodes have the same address in the computer's memory and the representation becomes strongly canonical. Thus, testing the equivalence of two Boolean functions

represented by OBDDs is reduced to a simple address pointer comparison. Note that every single node of a shared OBDD determines an OBDD of its own.
While working with OBDDs a fast access to distinct OBDDs, or sub-OBDDs is highly desirable. Therefore, most implementations rely on storing OBDD nodes in hash tables. The hash function, which determines the position of the node in the hash table can be computed from the addresses of the node's both successors and its variable assignment. At this position a pointer to the node's address or even the node itself is stored. Because it is possible that the hash function is mapping different nodes to the same position in the hash table, collision lists have to be maintained. If the node is directly stored in the hash table, each node requires a *next* pointer, connecting the nodes in the collision list. Most software packages for OBDDs are working with one separate hash table for each input variable x_i, which is important for the efficient implementation of dynamic minimization techniques (see *Minimization of OBDDs*). Thus, all nodes are stored in the hash table according to their variable assignment.
Before a new OBDD node is created, it can be tested, whether a node with the required properties does already exist. We simply have to compare all nodes in the collision list of the computed hash table slot, whether there exists a node of the same variable and with the same successors as the node that has to be created.

The ITE-algorithm and Caches

The task of transforming a description of a digital circuit, often given as a netlist of gates to an OBDD representation is called *symbolic simulation*. During this process for every gate starting from the inputs to the output gates of the circuit an OBDD is computed gradually from the OBDDs representing the gate's predecessors and the Boolean function realized by the gate. To connect two Boolean functions f and g given in terms of OBDDs with an arbitrary Boolean operation $\otimes$ the Boole-/Shannon decomposition w.r.t. variable x_i is applied:

$$f \otimes g = x_i(f|_{x_i} \otimes g|_{x_i}) + \overline{x_i}(f|_{\overline{x_i}} \otimes g|_{\overline{x_i}}).$$

The composition of the cofactors can be computed recursively. The recursion stops if one of its parameters is a constant or if the operation can be computed trivially. Note, that for each variable the number of computations doubles during this recursion. To avoid this exponential blow up it is mandatory to maintain a cache for storing already computed results.
For efficiency reasons all Boolean operations are mapped to a single general operation, which is able to express all Boolean operations, the so called *If-Then-Else operator (ITE)* [BRB90]. Then, we only have to maintain one single operator cache. $ITE(x, y, z)$ is a three parameter function computing *if x, then y, else z.*

$$ITE(x, y, z) = x \cdot y + \overline{x} \cdot z$$

Thus, the ITE operator refers exactly to the operation that is performed in an OBDD node and additionally, we do not have to provide different terminal cases for different operators. For computing the synthesis of functions f, g, h

represented as OBDDs, ITE is evaluated recursively w.r.t. the top variable of the involved OBDDs.

$$ITE(f,g,h) = (x_i, ITE(f|_{x_i}, g|_{x_i}, h|_{x_i}), ITE(f|_{\overline{x_i}}, g|_{\overline{x_i}}, h|_{\overline{x_i}}))$$

The recursion stops, if the first argument is constant, if the second and the third arguments are constant, or if the second and the third arguments are equal.
The operation cache for storing already computed results is called *computed table*. For every possible combination of subfunctions of f, g, h it contains an entry with the result computed by the application of $ITE(f,g,h)$. Now, it is obvious, why every binary Boolean operation is mapped to ITE. Otherwise, we would have to provide larger cache table entries also containing the operator itself, or we would have to maintain a distinct cache table for each operator. Thus, the mapping of each operation to a single operator increases the efficiency of the cache.
If the computed table already contains all computed partial results, then the time complexity of the ITE algorithm is limited by $O(|G_f| \cdot |G_g| \cdot |G_h|)$. For binary operations the run time is even only quadratic, because one of the three arguments is constant.
In most implementations the computed table is maintained as a hash based cache without using dynamically allocated collision lists, i.e. each slot of the hash table contains at most $k \in \mathbb{N}_0$ entries. If more than k entries are mapped to the same slot, an old entry will be overwritten. Experience has shown that most results are only reused shortly after their computation, which enables the efficient usage of a computed table with only small size.
To increase the hit rate of the computed table, the entry (f,g,h) is transformed into a standard form (f^*, g^*, h^*) via equivalences like e.g. $ite(f,f,g) = ite(f,1,g)$, $ite(f,g,f) = ite(f,g,0)$, etc. Also, if complemented edges are maintained (see the following chapter for more details), we have to take care that the creation of nodes with complemented 1-edge is avoided. Whenever a new node is created by the algorithm we have to test its existence via a lookup in the unique table.
See Fig. 2.4 for the ITE-algorithm in pseudo code.

Complemented Edges

In OBDDs the only difference between the function f and its complement $\overline{f}$ is the interchange of the two sink values 0 and 1. This similarity can be used by employing a single attribute bit for each edge. If this bit is set, the function denoted by the edge is taken as its complement. Thus, f and $\overline{f}$ can be represented by the same graph: $\overline{f}$ will be expressed by an edge pointing to the root node of f with the complement bit set to 1 [Ake78, MB88]. When using complemented edges we require only a single sink, the 1-sink. The Boolean constant 0 can be represented by the complement of 1. See Figure 2.5 for an example. Complemented edges are denoted by a dot on the arcs.

In most implementations OBDD nodes are identified by their memory address. For complementation of the function f represented by the OBDD P_f the least

```
Input: OBDDs f, g, h
Output: OBDD res representing res = ite(f,g,h).

ITE(f,g,h) {
  transform_to_standard_triple(f,g,h);
  if (res = terminal_case(f,g,h)) {
    return(res);
  }
  reorder_triple_acc_to_variable_order(f,g,h);
  check_rules_for_complemented_edges(f,g,h);
  if (res = in_computed_table(f,g,h)) {
    return(res);
  } else {
    x = top-variable(f,g,h);
    new_left=ITE(f_x, g_x, h_x);
    new_right=ITE(f_x̄, g_x̄, h_x̄);
    if (new_left==new_right) {
      res=new_left;
    } else {
      res=create_node(x,new_left,new_right);
    }
    insert_in_computed_table(f,g,h,res);
  }
  find_or_add_in_unique_table(res);
  return(res);
}
```

Figure 2.4: The ITE-Algorithm for OBDDs.

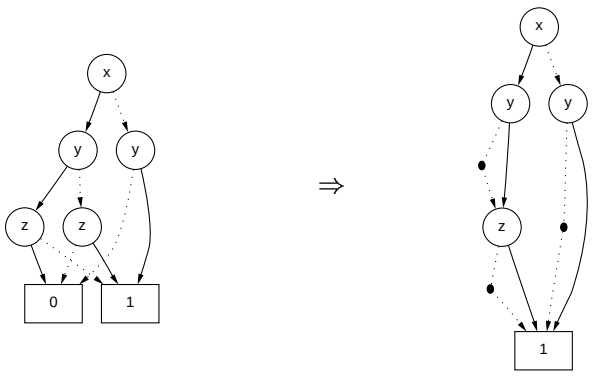


Figure 2.5: OBDD with Complemented Edges

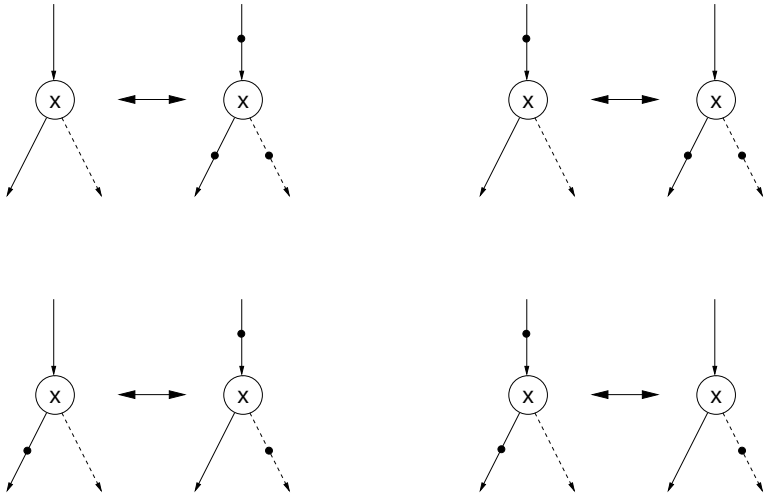


Figure 2.6: Equivalences for Complemented Edges

significant bit of the memory address is inverted. Because today for almost all computer architectures newly allocated pieces of memory are aligned to boundaries determined by the underlying word size, e.g. 32 or 64 bit, complemented OBDD nodes can easily be determined, because their least significant address bit is set to one, while for nodes that are not complemented the least significant bit is set to zero, respectively.
Unfortunately, one is losing canonicity when working with complemented edges. Therefore, we have to limit the usage of complemented edges to 0-edges, which can be assured by utilizing the following pairs of equivalences (see Figure 2.6).

The advantage of using complemented edges lies in the potential of an up to 50% reduction in size of the OBDD. Furthermore, the negation of a function can be computed in constant time, and the computation of Boolean operations can be accelerated by utilizing rules as $f \cdot \overline{f} = 0$ and $f + \overline{f} = 1$.
For working with OBDDs in practice, experience has shown that the reduction in size in most cases is limited to about 10%. But, the possibility of negating functions in constant time accelerates operations often up to a factor of 2.

Relevant Software Systems

In recent years different OBDD packages have been developed providing functions for the manipulation of switching functions. The first efficient implementation of the OBDD data structure dates back to 1990 and was developed by Brace, Rudell, and Bryant at the Carnegie Mellon University, Pittsburgh PA [BRB90]. The package is available to public and many of the implementation techniques described in this chapter have been developed in the context of this work. In 1993 Long [Lon93], also from Carnegie Mellon University introduced a new OBDD package that was mainly supposed to be utilized in symbolic model checking. This package for the first time was providing dynamic vari-

able reordering techniques for OBDD optimization and was included in the SIS software for synthesis of sequential systems [SSL+92].
The currently most developed open-source OBDD package is provided by the University of Colorado in Boulder and was developed by Somenzi in 1996 [Som96]. CUDD (Colorado University Decision Diagrams) is publicly available and has become the state-of-the-art package, which is constantly improved and maintained. Sophisticated implementation of algorithms as well as efficient memory management strategies are responsible for the great improvement in run time provided by CUDD. The CUDD programming package is also included in the VIS verification software of Berkeley [BHS+96].
The $\oplus$-OBDD package that has been developed based on the work of this thesis is adapting many of the algorithms and techniques that were introduced with CUDD.

2.1.3 Minimization of OBDDs

When working with OBDDs the available main memory of the computer is the most important limiting factor. If the main memory is not sufficient for storing an OBDD, secondary memory with much longer access time has to be utilized. But, manipulation of an OBDD that is partially stored in secondary memory is much too time consuming in practice. Thus, efficient minimization techniques for OBDDs are of the utmost importance to keep the representation as small as possible.
The size of an OBDD representation of a Boolean function crucially depends on the order of the input variables. In this section the importance of a well chosen variable order is shown and techniques how to improve the variable order dynamically based on a local variable exchange are introduced.

The Variable Order

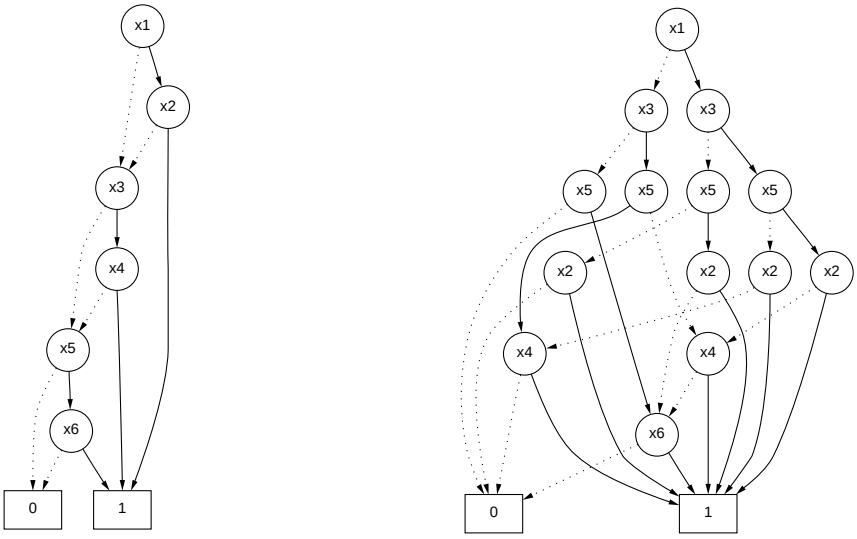
The size of an OBDD and thus, the complexity of its manipulation depends on the chosen order of the input variables. For example, the OBDD size of the disjunctive quadratic function

$$DQF_n(x_1, \ldots, x_{2n}) = x_1x_2 + x_3x_4 + \ldots + x_{2n-1}x_{2n}$$

is rather sensitive w.r.t. any change of the variable order. For the order $x_1, x_2, \ldots, x_{2n-1}x_{2n}$ the reduced OBDD consists of exactly $2n + 2$ nodes and thus, is of size linear in the number of input variables. But, for the variable order $x_1, x_3, \ldots, x_{2n-1}, x_2, x_4, \ldots, x_{2n}$ the OBDD consists out of 2^{n+1} nodes. The reduced OBDD for this variable order grows exponentially. See Figure 2.7 for an example of DQF_n for $n = 3$.

Other functions with high sensibility to the variable ordering resulting in OBDD sizes ranging from polynomial to exponential size are e.g.:

- the **direct storage access** function DSA_n, defined over $n + k$, $n = 2^k$ input variables $x_0, \ldots, x_{k-1}, y_0, \ldots, y_{n-1}$, computing $\mathrm{DSA}_n(x, y) = y_{|x|}$, where $|x|$ is the number whose binary representation equals x,



(a) variable order $x_1, x_2, \ldots, x_6$ (b) variable order $x_1, x_3, x_5, x_2, x_4, x_6$

Figure 2.7: The Function $DQF_3(x_1, x_2, x_3, x_4, x_5, x_6) = x_1x_2 + x_3x_4 + x_5x_6$.

- the **comparison** function COM that is deciding whether $|x| \geq |y|$ for two n-bit inputs $x = x_1 \ldots x_n$ and $y = y_1 \ldots y_n$, or
- the computation of the most significant bit in the **addition** of two binary encoded numbers.

But, because OBDDs are a canonical representation of Boolean functions it is not possible to find concise representations for all Boolean functions, i.e. representations of non exponential size w.r.t. the number of input variables. There are 2^{2^n} Boolean functions $f : \{0,1\}^n \to \{0,1\}$ and just because of this simple counting argument, there must be OBDDs for many functions that are of exponential size, no matter what ordering of the input variables is chosen [Sha49]. Although for that reason a randomly chosen Boolean function requires an exponentionally large representation, functions occurring in practice are usually strongly structured and have many inherent symmetries. These properties can be exploited by OBDDs and often lead to a succint representation.
For an example of Boolean functions that can only be represented by OBDDs of exponential size, see

- the **Hidden Weighted Bit** (HWB) function [Bry91]

$$HWB(x_1, \ldots, x_n) = \begin{cases} x_{wt(x_1,\ldots,x_n)} & \text{if } wt(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i > 0 \\ 0 & \text{otherwise,} \end{cases}$$

- the **multiplication** of two n-bit binary encoded numbers [Bry91],
- or the **Parity of 3-clique** function $\oplus$-$cl_3(x_1, \ldots, x_n)$ [ABH+86], deciding, whether the number of triangles in a given undirected graph is even or odd.

One possibility for achieving an appropriate variable order for an OBDD are heuristics based on the underlying circuit topology of the Boolean functions to be represented, if the function is given in terms of a netlist, representing a circuit. Starting from the outputs of the circuit description the circuit tree is traversed and the input gates are ordered according to their significance for each gate and hence, for the represented function [MWB88, FFK88, MIY90]. These approaches are often based on the idea that variables should be tested first, if they influence many subparts of the circuit. Moreover, variables which are close together in the circuit should also be close together in the variable ordering. Thus, *finding important variables* and *grouping together related variables* are the basic principles in these static approaches.
More sophisticated techniques like variable interleaving [FOH93] have to be applied, if functions with multiple outputs are represented by a shared OBDD. But, the power of these static approaches is rather limited and often, there exist much better variable orders for the functions under consideration.

Local Variable Exchange

Besides the static algorithms mentioned before, another approach for achieving better variable orders is based on the exchange of variables, which are adjacent according to the given variable order in an already existing OBDD representation. This operation is also often referred as the *swap*-operation. By using this *dynamic reordering* technique one is able to improve the variable order even during the construction of the OBDD. All dynamic reordering procedures are based on the fact that two adjacent variables of an OBDD within a given variable order can be exchanged efficiently [FMK91], and this basic operation remains local and does not have global effects on the entire OBDD.
Let the variable x_i be positioned directly before variable x_j w.r.t. the given variable order π, and let f be the function represented by a node labeled with x_i. Then, according to the Boole-/Shannon-decomposition the following proposition holds:

$$f = x_i x_j f_{11} + x_i \overline{x_j} f_{10} + \overline{x_i} x_j f_{01} + \overline{x_i}\,\overline{x_j} f_{00}.$$

If we reorder the terms while applying the rules of commutativity in the way that x_j will be positioned in front of x_i, then we obtain

$$f = x_j x_i f_{11} + x_j \overline{x_i} f_{01} + \overline{x_j} x_i f_{10} + \overline{x_j}\,\overline{x_i} f_{00}.$$

Thus, only the subfunctions f_{01} and f_{10} have to be exchanged in the OBDD (See Figure 2.8).

No other nodes in the OBDD, except those labeled with x_i and x_j are affected and thus, the swap-operation remains local.

Exact Algorithm and Heuristics

Due to the fact that the size of an OBDD crucially depends on the chosen variable order, algorithms for computing well suited orders are of significant

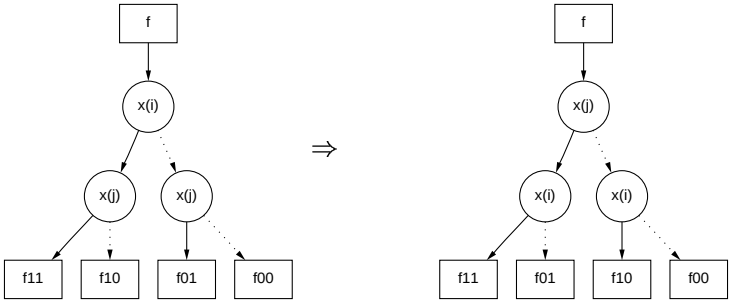


Figure 2.8: Exchange of Adjacent Variables

importance. But, there is no efficient algorithm for determining the best variable order among all possible variable orders. In 1996 Bollig and Wegener have shown that even the improvement of a given variable order remains **NP**-complete [BW96].
Exact optimization algorithms, like the one given in [FS90, ISY91] are based on *dynamic programming* or *branch and bound* methods, and have been further improved in 1998 by [DDG98]. But, because of the run time being exponential in the number of input variables, these methods are only suitable for circuits with a rather small number of input variables.
What makes the problem of finding the best variable order so hard is the fact that the only way of judging the quality of a given variable order is the explicit construction of the according OBDD. If the chosen variable order is not suited for the represented function, the OBDD to be constructed will be of exponential size. Thus, this optimization problem is much harder to solve than e.g. the well known *Traveling Salesman Problem* (TSP). In TSP the number of possible solutions, i.e. the number of possible round trips through n cities is equal to the number of possible variable orders of n variables, namely $n!$. But, the quality of the function to be optimized in TSP can simply be determined by the addition of n integers, while the construction of an OBDD may require exponential time and space.
Therefore, in a practical working environment, one has to apply heuristical methods for finding an appropriate variable order. Heuristics are reducing the search space - here the number of variable orders to be investigated - drastically, and thus, most times got stuck in some local but acceptable minimum. The heuristics for OBDDs are ranging from simple greedy methods [Rud93] to more sophisticated algorithms that are also taking *symmetries* [PS95] or *sampling* methods [MS97] under consideration.

The Sifting Algorithm

In 1993 Rudell proposed the so called *Sifting* algorithm for dynamic minimization of OBDDs [Rud93]. The algorithm is based on the application of a subroutine, which for a given variable is searching the optimal position without changing the position of the remaining variables. The variable is moved through every position in the variable order by ongoing exchange with its neighbor. This

```
Input: OBDD P_f, with variable order Π, and growth factor γ.
Output: OBDD P'_f, P'_f ≤ P_f with variable order Π'.

sifting (P, γ) {
   create ordered list of variables x_i, 1 ≤ i ≤ n;
   foreach variable x_i {
    repeat {
     move x_i through all levels j, 1 ≤ j ≤ n, while
     storing |P_j|, the size of P with x_i in level j
    }
    until (|P_j| > γ · |P| or all levels j have been accessed)
    target = level j with |P_j| = min(|P_i|),   1 ≤ i ≤ n;
    move x_i to level target;
   }
   return(P);
}
```

Figure 2.9: Outline of the Sifting Algorithm in Pseudo Code.

routine is subsequently called for all variables until it ends in some local minimum. During moving a single variable through the order the size of the OBDD may grow dramatically. Then, it is most unlikely that a local minimum will be reached in one of the following steps. Therefore, the search routine stops, if the growth of the OBDD is exceeding a given growth factor γ, $\gamma \in \mathbb{R}$. Also the order in which the single variables are accessed is important. Usually, the variables are ordered according to the number of nodes labeled with that variable. The variable that has the most nodes associated with, will be moved first, because there is a high probability that a change of the position of this variable will affect the OBDD size most. See Fig. 2.9 for a brief outline of the sifting algorithm in pseudo code.
The idea of sifting is up to now the most successful approach for the construction of good variable orders in a practical working environment. This stems due to the fact that it may move a variable fast over a long distance within the given order and that it is possible to leave a local maximum of size in search space again, because the position to which the variable is moved is only dependent of the minimum found and not of the in between lying local maxima.
Obviously, there is a given space-time trade off. The algorithm becomes faster, if we are using a limiting growth factor γ, but more possible solutions are explored for larger γ-values or even $\gamma = \infty$, because then, we are able to escape more often from local minima. Efficient implementations have to consider an appropriate choice of γ, the sequence in which the variables are chosen, and the direction in which the variables are moved first, what is depending on their position in the variable order. Also properties like *symmetries*, *interaction of variables* [PS95], and theoretical *lower bounds* [DG99] can be applied to speed up the sifting algorithm.

Chapter 3

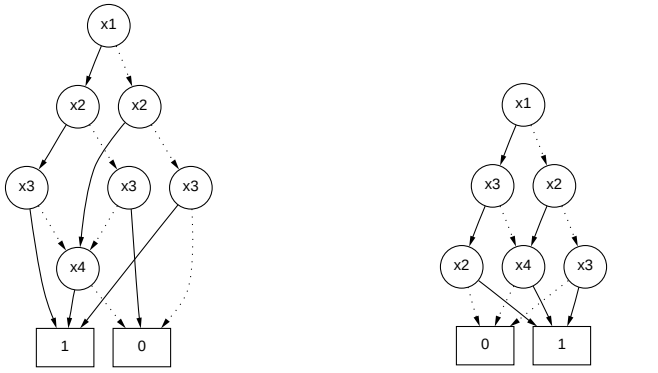
Extensions of OBDDs

In this chapter the question is investigated, whether OBDDs are sufficient as a data structure for canonically representing Boolean functions. Although most Boolean functions of practical interest can be represented efficiently with OBDDs, most Boolean functions must be of exponential OBDD size, no matter what variable order is chosen. To cope with functions of exponential OBDD size, one idea is to relax some of the restrictions that have been risen for OBDDs. Unfortunately one of the consequences might be the loss of canonicity and a dramatic increase in difficulty of the manipulation tasks. Some extensions of OBDDs based on the relaxation of given restrictions are presented and the main focus of this chapter lies on the introduction of so called *operator nodes* into the OBDD data structure. These operator nodes compute a binary Boolean operation from the functions that the successors' BDDs are computing. The best suited operations for all purposes are the Boolean *Exclusive Or* (EXOR, Parity,$\oplus$) and the Boolean *Equivalence* (EQU) and therefore, we are concentrating on $\oplus$-OBDDs, i.e. OBDDs extended with $\oplus$ operator nodes.

3.1 OBDDs - an Ideal Data Structure?

As already mentioned above, many Boolean functions of any practical relevance can be efficiently represented by OBDDs. But, this observation does not hold for every function, simply because of Shannon's counting argument cited in the previous chapter, there are also Boolean functions of importance for practice that can not be represented in a concise way by OBDDs, like e.g. the multiplication of two n-bit binary numbers [Bry91]. Unfortunately, this fact holds for all possible variable orders and thus, there is a need to think of different data structures for representing them. To make the OBDD data structure more powerful, one may think of relaxing the given restrictions for OBDDs. Thus, on the one hand giving this data structure the potential of being more concise, but on the other hand their manipulation becomes often much more difficult. One reason for that lies in the fact that canonicity of the data structure might get lost by losing certain restrictions. The single restrictions that are taken under consideration for relaxation are:

- the ordering restriction,



(a) OBDD (b) FBDD

Figure 3.1: Example of a FBDD (a) and an OBDD (b) for $f(x_1, x_2, x_3, x_4) = \overline{x_1}\overline{x_2}x_3 + \overline{x_1}x_2x_4 + x_1\overline{x_3}x_4 + x_1x_2x_3$

- the read once property,
- the internal node functionality in general, or
- the introduction of additional nodes with different functionality.

Other forms of extensions like the introduction of multiplicative or additive edge values [LPV94], or extensions to the word level domain using momentum based function decomposition [BC95] are not further considered in this work. The question to be investigated is:

> *Is it possible to transform OBDDs to a data structure being more concise, while simultaneously maintaining their nice algorithmic properties?*

3.2 Extensions of OBDDs

3.2.1 Free BDDs and Read-k Decision Diagrams

If we give up the ordering restriction for OBDDs completely, we end up in *read once branching programs*, also called *Free Binary Decision Diagrams*.

Definition 3.1 *A BDD is called a* **free binary decision diagram** *(FBDD) if, on each path, each variable is tested at most once.*

See Figure 3.1 for an example.

The size of a minimal FBDD for a given Boolean function is, of course at most the size of a minimal OBDD, but in many cases, it can be exponentially smaller. For example the *indirect storage access function* or the *hidden weighted bit function* can be represented in quadratic FBDD size, while OBDD representations of these functions are always of exponential size. Unfortunately, operations

like Boolean synthesis of two FBDDs are NP-hard [GM94b] but, testing the equivalence of two FBDDs can be decided probabilistically in polynomial time [BCW80]. Testing satisfiability for FBDDs can be realized in time linear to the size of the FBDD by simply testing in a *depth-first-search*-approach (dfs), whether the 1-sink is reachable on any path starting at the root node. Function evaluation, testing satisfiability, and replacement of variables by Boolean constants are of equal complexity as for OBDDs.
But, the problem of FBDD synthesis being NP-hard remains and thus, for working more efficiently with FBDDs, the concept of *FBDD types* was introduced by Gergov and Meinel [GM94a], and independently by Sieling and Wegener [SW95]. A FBDD type τ is a generalization of the linear order for OBDDs and is defined like an FBDD with only one sink. In a FBDD F_τ related to a specific type τ all variables are tested according to the graph given by τ. Then, for two FBDDs F_τ and G_τ, both related to type τ, binary Boolean operations can be computed efficiently. Typed FBDDs can be reduced in the same way as OBDDs. Then, FBDDs related to a specific completely reduced type τ^* are a canonical representation for Boolean functions.
But, the size of a FBDD crucially depends on the chosen type, and generating a well suited type for FBDD operations is a task that is even more difficult than generating a well suited linear order for OBDDs. Another disadvantage in the case of FBDDs is that in contrary to OBDDs in the lower levels of the graph there is less possibility of sharing between subfunctions, because of the different variable orders of the FBDDs representing these subfunctions. Because of this particular property, it is often not possible to compute FBDDs that are significantly smaller than the OBDD representation of the same function.
By relaxing not only the ordering restriction, but also the read once property, we will get a binary decision diagram, where on each path, every variable may occur several times in a random order. BDDs, where each variable appears on every path at most k times, $k \in \mathbb{N}$, are called *read-k-times-only branching programs* (k-BP).

Definition 3.2 *Let $k \in \mathbb{N}$. A read-k-times decision diagram (read-k-times branching program, k-BP) is a depth restricted decision diagram, where each variable appears on every path from the root to the sink at most k times.*

Thus, FBDDs as a special case of k-BPs, can be regarded as 1-BPs. In contrast to FBDDs, k-BPs may contain inconsistent paths (also called *null-chains*). An inconsistent path is a path, which cannot be part of a computation path, since at least one variable would have to be tested with different results on it.
Several variants of k-BPs have been proposed, like e.g. *k-indexed binary decision diagrams* (k-IBDDs) [JAB+92], where the decision diagram is divided into k layers, each layer obeying a variable order π_i, $1 \leq i \leq k$, or *k ordered binary decision diagrams* (k-OBDDs)[BSS+98], where each layer has the same variable order $\pi_i = \pi$, $\forall i$. But, although k-BPs or restricted models of k-BPs ($k > 1$) have the potential of being more concise than FBDDs or OBDDs [Weg87], they are even more difficult to manipulate [CHS74]. Synthesis of two k-BPs might cause an exponential blow up in size, testing the equivalence of two k-BPs is

(a) OBDD (b) OFDD

Figure 3.2: OBDD Node and OFDD Node

known to be *co-NP-complete* and k-BP-satisfiability is *NP-complete* even for 2-BPs [FHS78] or 2-IBDDs.

3.2.2 Functional Decision Diagrams

Another way of extending the concept of OBDDs is changing the way a function is computed in the single branching nodes. In OBDD nodes the function is computed according to the Boole/Shannon decomposition (BS) w.r.t. the variable x_i that the node is labeled with: $f = x_i f_{x_i} + \overline{x}_i f_{\overline{x}_i}$. The advantage of this functionality lies in the fact that each input activates exactly one path, called its computation path. In some applications one works with representations of Boolean functions by $\mathbb{Z}$-polynomials, i.e. $\oplus$-sums of monomials of positive literals. This representation is related to Reed-Muller's expansion rule (RME) [Ree54, Mul54], also often referred to as the *positive Davio expansion* (pDE), or, if we consider only monomials of negative polarity, the *negative Davio expansion* (nDE) [BD95].

$$\begin{aligned} \text{pDE: } f &= f|_{\overline{x}_i} \oplus x_i(f|_{x_i} \oplus f|_{\overline{x}_i}) \\ \text{nDE: } f &= f|_{x_i} \oplus \overline{x}_i(f|_{x_i} \oplus f|_{\overline{x}_i}). \end{aligned}$$

The correctness of these decompositions follows by the consideration of the cases $x_i = 0$ and $x_i = 1$. The decompositions are unique.
If $f = g \oplus x_i \cdot h$ for arbitrary Boolean functions f and g that are not essentially depending on x_i, then $f|_{x_i=0} = g$ and $f|_{x_i=1} = g \oplus h$, and therefore, $h = f|_{x_i=0} \oplus f|_{x_1=1}$. This motivated Kebschul, Schubert and Rosenstiel to introduce *Ordered Functional Decision Diagrams* [KSR92], which are not based on the Boole/Shannon decomposition as OBDDs, but on the positive Davio expansion (see Figure 3.2).

Definition 3.3 *Ordered Functional Decision Diagrams (OFDDs) are defined in the same way as OBDDs, but the function f_v that is computed by an OFDD node v is based on the following inductive rules:*

1. *If v is a sink labeled with 1 (0), then $f_v = 1$ ($f_v = 0$).*
2. *If v is a node labeled with variable x_i, whose successor nodes are representing the functions g and h, then $f_v = g \oplus x_i h$.*

Note that an input $a \in \{0,1\}^n$ can activate more than a single path on an OFDD. At a node v labeled with x_i and the input $a_i = 0$, it is sufficient to consider the 0-successor and only the 0-edge is activated. But, if $a_i = 1$ we have to consider both successors and to take the $\oplus$-sum of their results. Then, both outgoing edges are activated. An input a that contains j 1s activates 2^j edges in a complete OFDD.
OFDDs can be reduced similar to OBDDs: if the 1-successor of an OFDD node is pointing to the 0-sink, the node is redundant and can be removed while redirecting all incoming edges to its 0-successor. Isomorphic subgraphs can be reduced in the same way as for OBDDs. If no reduction rule can be applied to an OFDD, then the OFDD is completely reduced. As reduced OBDDs, reduced OFDDs are a canonical representation for Boolean functions.
A point that may cause problems for working with OFDDs is that the replacement of a variable x_i in an OFDD F by a constant $c \in \{0,1\}$, which is an essential *low level* operation for all other manipulation tasks, is much more difficult than for OBDDs. The best known algorithm for replacement of variables by constants for OFDDs has running time $\Theta(|F|^3)$, and already logarithmically many applications of this operation may increase the size of the representation exponentially [BLS+95].
For some classes of functions OFDDs are exponentially more compact than OBDDs but, for other classes of functions the opposite holds. Because of this observation for combining the advantages of both decision diagram types a hybrid representation called *Ordered Kronecker Functional Decision Diagram* (OKFDD) was introduced [DST+94]. Each variable x_i in an OKFDD has an assigned decomposition type $d_i \in \{BS, pDE, nDE\}$. In every node labeled with variable x_i the decomposition d_i is computed. For a fixed decomposition type order OKFDDs are also a canonical representation for Boolean functions. The list of possible decompositions of Boolean functions is limited to the three decompositions pDE, nDE, and BS, if we demand that the functions represented at the successors v_0, v_1 of a node v labeled with x_i do not essentially depend on x_i and that the function f_v represented by v can be computed by some operations $op : \{0,1\}^3 \to \{0,1\}$ from x_i, v_0, and v_1. Additionally, we don't distinguish operations leading to an isomorphic graph structure [BD95].

3.2.3 Binary Decision Diagrams with Operator Nodes

When changing the functionality of OBDD nodes the next step may be the introduction of additional operator nodes into the data structure, i.e. nodes that are not labeled with a Boolean variable but only with a binary Boolean operation $\omega \in \Omega = \{\wedge, \vee, \oplus, \equiv, \ldots\}$. This operator node computes a function f resulting from the application of its operator ω to its two successors f_1, f_2, $f = f_1 \omega f_2$. *Ω-Branching Programs* introduced by Meinel in [Mei88] are generalizing this concept.

Definition 3.4 *Let Ω be a set of binary Boolean operations. An Ω-branching program (Ω-BP) on the variable set $X = \{x_1, \ldots, x_n\}$ is a directed, acyclic graph with the same structure as a regular branching program, but which may*

additionally contain nodes labeled with a function $\omega \in \Omega$ instead of a variable.

$\vee$-branching programs are also known as *non-deterministic* branching programs.

Definition 3.5 *A non-deterministic branching program is a directed, acyclic graph with the same structure as a regular branching program, but which may additionally contain binary nodes labeled with the operation $\vee$ ($\vee$-nodes). An assignment $a = (a_1, \ldots, a_n)$ to the input variables x_i, $1 \leq i \leq n$ computes 1 if there exists a path compatible to a, i.e. at a branching node x_i the edge a_i is chosen and at a $\vee$-node an arbitrary edge can be chosen.*

$\wedge$-branching programs are also called *co-non-deterministic branching programs* and $\{\vee, \wedge\}$-branching programs are called *alternating non deterministic branching programs*. Unrestricted $\{\vee, \wedge\}$-branching programs can be considered to be similar to circuits that are based on $\{\vee, \wedge, \neg\}$, since they are polynomially related to each other. Although being a rather compact data structure for representing Boolean functions, manipulation tasks cannot be carried out efficiently. Despite this fact, $\{\vee, \wedge\}$-branching programs have also been considered under the name *XBDDs* [JPH+91]. The idea is to use as "few non-determinism" as possible. XBDDs are restricted by splitting the branching program in an upper and a lower part. The upper part consists of nodes labeled only with $\exists$ and $\forall$ (which corresponds to $\vee$ and $\wedge$), and the lower part consists of OBDD like structures. But, XBDDs as well as $\{\vee, \wedge\}$-branching programs have the obvious drawback that testing satisfiability is NP-complete.
In [Mei90] it was shown that, within polynomial size, each Ω-BP is computationally equivalent to an Ω'-BP, $\Omega' \in \{\{\vee\}, \{\wedge\}, \{\oplus\}, \{\vee, \wedge\}\}$.
If we restrict Ω-BPs to Ω-BP1s or Ω-OBDDs, manipulation tasks become easier, but the drawback of being non-canonical representations for Boolean functions remains. Testing equivalence of ω-OBDDs, $\omega \in \{\{\wedge\}, \{\vee\}, \{\wedge, \vee\}\}$, is **co-NP-complete**, if it is computed deterministically. Only for $\omega \in \{\oplus, \equiv\}$ testing the equivalence is within **co-RP**. The reason for this is explained later in detail in the section *probabilistic equivalence test.*
In fact, working with non canonical representations of Boolean functions necessitates the existence of an efficient equivalence test. In our case this fast equivalence test can only be performed with probabilistic techniques, i.e. we are generating hash codes identifying the instances of the data structure under consideration. But, the proposed hashing technique is only working in an efficient way, if we admit only the operators $\oplus$ or $\equiv$. The usage of conjunction ($\vee$) or disjunction ($\wedge$) as an operator node would require that the functions that are combined are of disjoint support for applying the probabilistic equivalence test. Thus, the subject of this work is the development of efficient manipulation techniques for $\oplus$-OBDDs.

Chapter 4

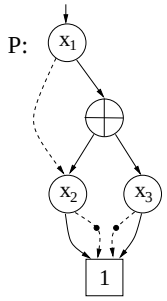
$\oplus$-OBDDs

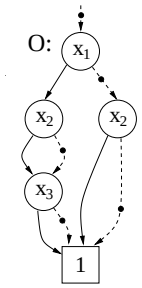
In this chapter the data structure for $\oplus$-OBDDs will be defined. After focusing on the properties of $\oplus$-OBDDs we will separate them from other derivatives of OBDDs and EXOR based data structures by comparing their algorithmic capacities. The well-known reduction concept for OBDDs will be expanded to be applicable for $\oplus$-OBDDs. But, for working efficiently with this non-canonical data structure, we are in need for a fast equivalence test. The fastest known deterministic equivalence test, based on a slightly changed version of the $\oplus$-OBDD model concept requires time cubic in the number of nodes, what makes it not suitable for an application in any commercial environment. Therefore, we consider a probabilistic equivalence test, which is based on an arithmetic transformation of the Boolean function to be represented by $\oplus$-OBDDs into a polynomial over a finite domain. Next, we proceed with the definition of algorithms for efficient $\oplus$-OBDD synthesis. For this task, the computation of the function's cofactors has to be adapted to the new data structure, as well as already known efficient synthesis procedures like the ITE algorithm. One problem remains, which is the way of how to introduce $\oplus$-nodes into the data structure during the synthesis procedure. We are adapting the ITE algorithm and the APPLY algorithm for $\oplus$-OBDDs with the help of alternative function decompositions (pDE, nDE), which provide explicit EXOR operations that can be directly mapped into $\oplus$-nodes. Finally, complexity results for the basic manipulation tasks for $\oplus$-OBDDs are summarized.

4.1 Definitions

If the Boolean operator $\oplus$ (XOR, EXOR) is chosen to serve as an operator node for extending the OBDD data structure we obtain so called $\oplus$-OBDDs (Parity-OBDDs, Mod2-OBDDs).

Definition 4.1 *A $\oplus$-OBDD P over a set $X_n = \{x_1, \ldots, x_n\}$ of Boolean variables is a directed acyclic connected graph $P = (V, E)$. V is the set of nodes, consisting of non-terminal nodes with out-degree 2, and of terminal nodes with out-degree 0. There is a distinguished non-terminal node, the root, which, as only node, has the in-degree 0. The two terminal nodes with no outgoing arcs*





x_1	x_2	x_3	f_P
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Figure 4.1: $\oplus$-OBDD P and OBDD O with Complemented Edges, Both Computing the Boolean Function f_P

are labeled with the Boolean constants 0 and 1. The remaining nodes are either labeled with Boolean variables $x_i \in X_n$, denoted as branching nodes*, or with the binary Boolean operator $\oplus$ (EXOR, Parity), denoted as* $\oplus$-nodes*. On each path, every variable must occur at most once. In the following, let $l(v)$ denote the label of the node $v \in V$ and $|P|$ the number of non terminal nodes of P.*
$E \subseteq V \times V$ denotes the set of edges. The two edges starting in a branching node v are labeled with 0 and 1. The 0(1)-successor *of node v is denoted by $v_0(v_1)$. There is a permutation π, which defines an order $x_{\pi(1)} < x_{\pi(2)} < \ldots < x_{\pi(n)}$ on the set of input variables. If w is a successor of v in P with $l(v), l(w) \in X_n$, then $l(v) < l(w)$ according to π must hold.*

Note that since the $\oplus$-operation is symmetric, the outgoing edges of $\oplus$-nodes do not have to be labeled separately. For representing multiple output Boolean functions $f : \{0,1\}^n \rightarrow \{0,1\}^m$, we consider *multi-rooted shared* $\oplus$-OBDDs by introducing multiple roots into a single $\oplus$-OBDD, each root representing a subfunction of $f = (f_1, \ldots, f_m)$, $f_i : \{0,1\}^l \rightarrow \{0,1\}$, $l \leq n$.
The function f_P associated with the $\oplus$-OBDD P can be evaluated in the following way: For a given input assignment $(a_1, \ldots, a_n) \in \{0,1\}^n$, the Boolean values assigned to the leaves extend to Boolean values associated with all nodes v of P as follows:

- Let v_0 and v_1 be the successor nodes of v, computing the Boolean values $\delta_0, \delta_1 \in \{0,1\}$.
- If v is a branching node labeled with $x_i \in X_n$, then v will be associated with δ_{a_i}.
- If v is a $\oplus$-node, then v will be associated with $\oplus(\delta_0, \delta_1) = (\delta_0 + \delta_1) \bmod 2$.

$f_P(a_1, \ldots, a_n)$ is associated with the value that is computed by the root node of P. Thus, the value of a Boolean function f_P for a given variable assignment represented by the $\oplus$-OBDD P can be evaluated in time $O(|P|)$.
Furthermore, we can also consider the use of complemented edges as already introduced for OBDDs in chapter 3 to achieve a more compact representation. See Fig. 4.1 for an illustrating example.

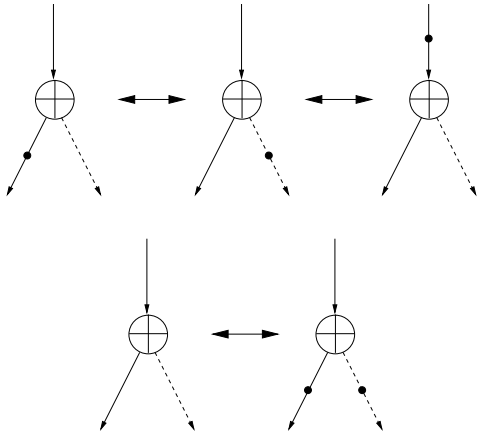


Figure 4.2: Equivalences for Complemented Edges and $\oplus$-Nodes.

In the example, let $f_P : \{0,1\}^3 \to \{0,1\}$ be defined by the given truth table. Moreover, let π be the natural order on the set of variables, i.e., $\pi(i) = i$. For branching nodes, the dashed line always represents the edge labeled with 0. A dot on an edge denotes that the edge is complemented and that it points to the Boolean complement of the function that is represented by its successor. Note that the edge pointing to the root of O is complemented, too.
As for branching nodes in regular OBDDs (see Fig. 2.6), in $\oplus$-OBDDs there is a set of equivalences for $\oplus$-nodes and complemented edges that has to be considered. In contrary as for OBDDs, restricting any newly created node to one distinct case of these equivalences for reasons of canonicity, here, this restriction is useful for avoiding further ambiguity, because the data structure is not canonical, anyhow. Additionally, this restriction leads to an increase in cache efficiency of the $\oplus$-OBDD implementation. See Fig. 4.2 for the complement equivalences concerning $\oplus$-nodes.

4.2 Properties of $\oplus$-OBDDs

Since OBDDs are special cases of $\oplus$-OBDDs (namely $\oplus$-OBDDs without any $\oplus$-node) and since, for each variable ordering, OBDDs provide a universal representation schema for Boolean functions, every Boolean function can be represented by means of a $\oplus$-OBDD.

Theorem 4.1 *Let π be an ordering on $X_n = (x_1, \ldots, x_n)$. Each Boolean function $f : \{0,1\}^n \to \{0,1\}$ over X_n can be represented by means of a $\oplus$-OBDD that tests variables according to π.*

Furthermore, from taking the assumption that each OBDD is also a $\oplus$-OBDD we can also conclude the following theorem.

Theorem 4.2 *The size of a minimal $\oplus$-OBDD for a given Boolean function f is at most the size of a minimal OBDD for f.*

To confirm the assumption that $\oplus$-OBDDs have the potential of being more concise than OBDDs are, we can show that there exist Boolean functions with small (low polynomial degree) $\oplus$-OBDD representation, whose OBDD representations have exponential size independent of the chosen variable order.
The first example is the *hidden weighted bit function HWB* as a very simple version of the storage access function: for simplicity, let x_0 denote 0 and let $wt(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i$. Here, wt is denoting the address of the bit that we are computing. Thus, *HWB* is defined by $HWB(x_1, \ldots, x_n) = x_{wt(x_1,\ldots,x_n)}$. *HWB* has the feature of an indirect storage access function. The whole input serves as indirect address, which is computed as the weight (sum) of the input. This weight is the direct address of the output bit. Intuitively, *HWB* is a very simple function. But, in [Bry91] it has been shown that for every possible variable order π each OBDD representation of the *HWB* must be of exponential size. One reason for the difficulty of *HWB* might lay in the fact that the complete input $(x_1, \ldots, x_n)$ serves as *control information* and simultaneously also as *data information*. But, Gergov and Meinel showed that the $\oplus$-OBDD representation of *HWB* requires only cubic size [GM96]:

Theorem 4.3 *The function $HWB(x_1, \ldots, x_n)$ can be represented with a $\oplus$-OBDD of size $O(n^3)$.*

Proof: The equation $HWB(x_1, \ldots, x_n) = \bigoplus_{k=1}^{n}(x_k \wedge E_k(x_1, \ldots, x_n))$, where $E_k(x_1, \ldots, x_n)$ computes 1, if the input assignment of $(x_1, \ldots, x_n)$ contains exactly k 1s can be verified easily. For each variable order, $x_k \wedge E_k(x_1, \ldots, x_n)$ can be represented with an OBDD of at most quadratic size, because the function $E_k(x_1, \ldots, x_n)$ is symmetric, what enables a quadratic OBDD representation for every variable order. For the representation of $x_k \wedge E_k(x_1, \ldots, x_n)$, every 0-edge of a node in the OBDD representing $E_k(x_1, \ldots, x_n)$ that is labeled with the variable x_k has to point to the 0-sink. Thus, the above equation can be immediately transformed into a cubic size $\oplus$-OBDD for $HWB(x_1, \ldots, x_n)$. □

For a survey on the complexity of *HWB* for various BDD models, see [BLS+99].
We can also easily show that a Boolean function represented by a $\oplus$-OBDDs is at most of the size of an *EXOR Sum of Products expression* (ESOP) representation of the same function.

Theorem 4.4 *The size of a minimal $\oplus$-OBDD for a given Boolean function f is at most the size of a minimal ESOP for f.*

Proof: With respect to a given variable ordering π, the monomials of an ESOP can be represented by OBDDs of size equal to the monomials. The EXORs of the ESOP can be simulated by $\oplus$-nodes and thus, any function represented by an ESOP can be represented by a $\oplus$-OBDD of at most the same size. □

However, in general the available reduction algorithm for $\oplus$-OBDDs that are simulating the ESOP expressions are able to provide even smaller representations. But, we can also proof that $\oplus$-OBDDs in fact provide much smaller representation sizes than ESOPs.

For an example of a Boolean function with exponential ESOP representation size, but small $\oplus$-OBDD representation size we chose the *majority function* MAJ that outputs 1, if the number of input bits with value one is greater than the number of input bits with value 0:

$$MAJ(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} x_i > \lfloor n/2 \rfloor \\ 0 & \text{else.} \end{cases}$$

In [Raz87] it has been shown that any representation of MAJ by means of multilevel $\{\wedge, \oplus\}$-expression (unbounded fan in and constant depth) has to be necessarily of exponential size. However, because MAJ is a symmetric function, for each variable ordering we can construct a $\oplus$-OBDD (indeed an OBDD) of quadratic size that is counting the number of ones. Thus, MAJ can be represented efficiently with polynomial size $\oplus$-OBDDs.
As an additional step, if we combine $MAJ(x_1, \ldots, x_n)$ and $HWB(y_1, \ldots, y_n)$ to the resulting function $f(x_1, \ldots, y_n) = MAJ(x_1, \ldots, x_n) \oplus HWB(y_1, \ldots, y_n)$, we have constructed an example of a function of small $\oplus$-OBDD representation size that on the other side necessarily has exponential representation size in terms of ESOPs as well as in terms of OBDDs. Therefore, we are able to conclude that $\oplus$-OBDDs can be considered to be a more compact data structure for the representation of Boolean functions than OBDDs or ESOPs are.
Also, the function $\oplus - cl_{n,3}(x)$, $x = (x_{i,j})_{1 \le i < j \le n}$, which decides whether a given undirected n-node graph $G = G(x)$ contains an odd number of triangles, has a $\oplus$-OBDD of size $O(n^3)$ [Mei88], although it is even for a powerful representation as FBDDs always of exponential size independently of the chosen variable order [ABH+86].

Theorem 4.5 *[Mei88] The function $\oplus - cl_{n,3}(x)$, $x = (x_{i,j})_{1 \le i < j \le n}$ can be represented by a $\oplus$-OBDD of size $O(n^3)$.*

Proof: Obviously, the OBDD P_{uvw}, $1 \le u < v < w \le n$, given in Fig. 4.3 decides, whether the three nodes of a given undirected graph $G = G((x_{i,j})$, $1 \le i < j \le n$ constitute a triangle. There are $\binom{n}{3}$ sets $\{u, v, w\} \subseteq \{1, \ldots, n\}$ with $1 \le u < v < w \le n$. If we are replacing all leaves of a binary tree of $\oplus$-nodes of size $2 \cdot \binom{n}{3} - 1$ with all possible OBDDs P_{uvw}, $u, v, w \in \{1, \ldots, n\}$, $1 \le u < v < w \le n$, then, we obtain a $\oplus$-OBDD of size $O(n^3)$, which computes $\oplus - cl_{n,3}(x)$. □

Thus, we have shown that $\oplus$-OBDDs also have the potential of being a more concise representation for Boolean functions than FBDDs are. In the next step, we compare $\oplus$-OBDDs with OFDDs, a representation for Boolean functions that is also depending on the EXOR-function (see Chapter 3). We show, how OFDDs can be simulated efficiently with $\oplus$-OBDDs and moreover, that there also exist exponential gaps between the two representation forms.

Theorem 4.6 *Let P_f be a OFDD that is representing an arbitrary Boolean function $f : \{0,1\}^n \to \{0,1\}$. Then, f can be represented by a $\oplus$-OBDD Q_f of size $|Q_f| \le 2 \cdot |P_f|$.*

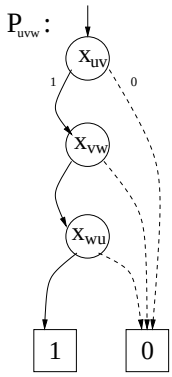


Figure 4.3: OBDD P_{uvw}, $u, v, w \in \{1, \ldots, n\}$, $1 \leq u < v < w \leq n$ for Deciding Whether uvw is a Triangle.

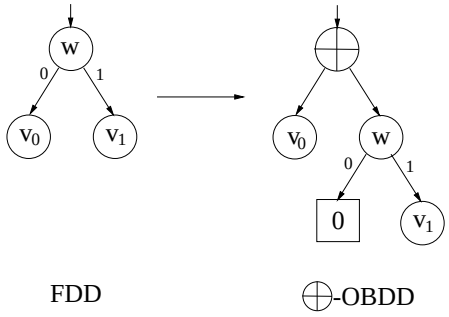


Figure 4.4: Transformation of a OFDD into a $\oplus$-OBDD.

Proof: Let us assume that each variable $x_i \in \{x_1, \ldots, x_n\}$ occurs in each path of P_f (i.e. P_f is not reduced). The sink of a source-to-sink path p of P_f is labeled with 1, if the monom that consists exactly of all variables that are positively tested on p appears in the pDE of f. A $\oplus$-representation of P_f can be created in the following way: Bottom up, for each node v of P_f we introduce a new node w also labeled with $l(v)$ and take the 1-successor v_1 of v as the 1-successor of w and the 0-sink as the 0-successor of w. Then, we are changing the node v to a $\oplus$-node and take the node w for its new second successor in addition to v_0 (see Fig. 4.4). □

In the last theorem we have shown, how OFDDs can be efficiently simulated with $\oplus$-OBDDs. On the other hand, Becker, Drechsler, and Theobald have shown that there are polynomial sized OBDDs for Boolean functions that can only be represented by exponential sized OFDDs [BDT97]. In particular they were showing that the function $1 - cl_{n,3}$, which is testing for the existence of exactly one single triangle in a given undirected graph, can be represented by OBDDs of size $O(n^4)$ [Weg00], but requires an OFDD representation of size $2^{\Omega(n)}$. In their proof the fact is used that the graph of the OFDD representation can also be interpreted as an OBDD. Thus, the OBDD P representing

Model	Function	Model size	$\oplus$-OBDD size
OBDD	$HWB(x_1, \ldots, x_n)$	$O(2^n)$	$O(n^3)$
FBDD	$\oplus$-$cl_{n,3}(x_1, \ldots, x_n)$	$O(2^n)$	$O(n^3)$
ESOP	$MAJ(x_1, \ldots, x_n)$	$O(2^n)$	$O(n^2)$
OFDD	1-$cl_{n,3}(x_1, \ldots, x_n)$	$2^{\Omega(n)}$	$O(n^4)$
	$HWB(x_1, \ldots, x_n)$	$2^{\Omega(n)}$	$O(n^3)$

Table 4.1: Exponential Gaps between $\oplus$-OBDDs and other Representations of Boolean Functions.

1-$cl_{n,3}$ is of size $O(n^4)$, while interpreted as an OFDD, the graph of P is computing the function $\oplus$-$cl_{n,3}$. Vice versa, the OFDD representation of 1-$cl_{n,3}$ must be isomorphic to the OBDD representation of $\oplus$-$cl_{n,3}$ and thus, must be of exponential size, while, on the other hand, the OBDD size of 1-$cl_{n,3}$, and thus, the $\oplus$-OBDD size is only polynomial.
Another example for an exponential gap between $\oplus$-OBDDs and OFDDs is the already mentioned HWB-function. In [BDW95] it was shown that the OFDD representation of HWB requires size $2^{\Omega(n)}$, while the $\oplus$-OBDD representation of HWB is only of size $O(n^3)$.
Thus, we may conclude that there exist $\oplus$-OBDDs that can only be represented by exponential sized OFDDs, and furthermore, that $\oplus$-OBDDs are a more concise representation of Boolean functions than OFDDs are.
To summarize the results of this section we may state that the representation forms of OBDDs, FBDDs, ESOPs, and OFDDs can efficiently be simulated in terms of $\oplus$-OBDDs. Simultaneously, there are exponential gaps between these standard representation forms and $\oplus$-OBDDs, confirming that $\oplus$-OBDDs are really a more powerful data structure. See table 4.1 for a summary of these results.

4.3 Reduction of $\oplus$-OBDDs

For working efficiently with $\oplus$-OBDDs, also the reduction concept introduced for OBDDs has to be adapted and extended. In general, the reduction rules for OBDDs are also suitable for $\oplus$-OBDDs, i.e. the *deletion rule* and the *merging rule* can also be applied to regular branching nodes within the $\oplus$-OBDD. For $\oplus$-nodes the rule set has to be extended accordingly.
First, the task of the deletion rule for branching nodes is to get rid of redundant nodes, i.e. branching nodes, which are connected to two identical successors. If the edges of a $\oplus$-node v happen to point to the same successor v_1, then the operation $f_v = f_{v_1} \oplus f_{v_1}$ has to be computed, which results in $f_v = f_{v_1} \oplus f_{v_1} = 0$. Thus, a $\oplus$-node with identical successors has to be replaced by the 0-sink.
Furthermore, the deletion rule set has to be extended for the use of *complemented edges*. If the edges of a $\oplus$-node v happen to point to the successors v_1 and v_0, where $f_{v_1} = \overline{f_{v_0}}$, then the operation $f_v = f_{v_0} \oplus f_{v_1}$ has to be computed, which results in $f_v = f_{v_0} \oplus \overline{f_{v_0}} = 1$. Thus, a $\oplus$-node with complementary

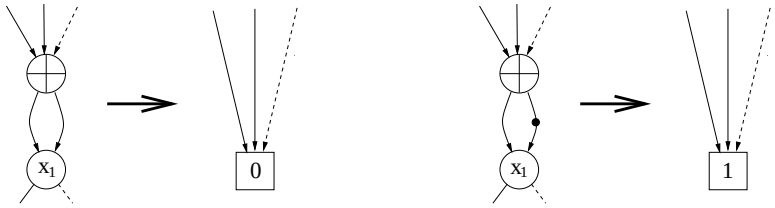


Figure 4.5: Deletion Rule Set for $\oplus$-Nodes

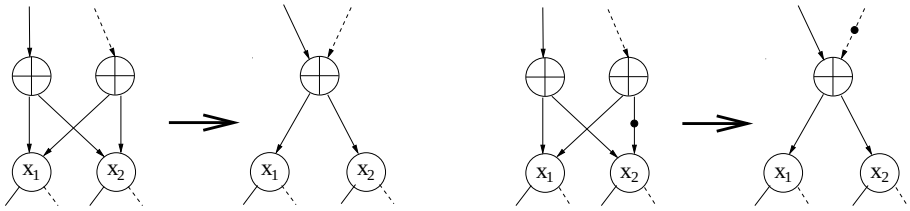


Figure 4.6: Merging Rule Set for $\oplus$-Nodes.

successors has to be replaced by the 1-sink (see Fig. 4.5).

For the application of the merging rule, isomorphic subgraphs have to be detected and identified. This rule is working for branching nodes as well as for $\oplus$-nodes in the same way. If we consider two $\oplus$-nodes v and w that are of different complementation parity, i.e. for the successors v_1, v_0 and w_1, w_0 the following holds: ($f_{v_1} = f_{w_1}$ and $f_{v_0} = \overline{f_{w_0}}$) or ($f_{v_1} = f_{\overline{w_1}}$ and $f_{v_0} = f_{w_0}$), then $f_v = \overline{f_w}$, and the two nodes can be identified, while regarding the rules for complementation (see Fig. 4.6).
In addition to the already introduced reduction rules for $\oplus$-OBDDs, we also have to consider the case that a successor of a $\oplus$-node is a terminal node. Then, the represented function can directly be computed: If a successor of a $\oplus$-node v, w.l.o.g let's take f_{v_1}, is the *0-sink*, then $f_v = f_{v_0} \oplus f_{v_1} = f_{v_0} \oplus 0 = f_{v_0}$ is computed and the $\oplus$-node is replaced by its other successor v_0. On the other hand, if the *1-sink* is a successor (f_{v_1}) of the $\oplus$-node, then $f_v = f_{v_0} \oplus f_{v_1} = f_{v_0} \oplus 1 = \overline{f_{v_0}}$ is computed and v is replaced by complementing all its incoming edges and connecting them to its second successor v_0 (see Fig. 4.7).

Additionally, there exists another reduction that is especially addressing $\oplus$-OBDDs of the following structure: Consider a $\oplus$-OBDD with a $\oplus$-node v at the top, representing the Boolean function f_v, and with the two successors v_0 and v_1, both labeled with the same variable $l(v_0) = l(v_1) = x_i$. Then, because the functions represented by the two successors of f_{v_0} and f_{v_1}, let them be

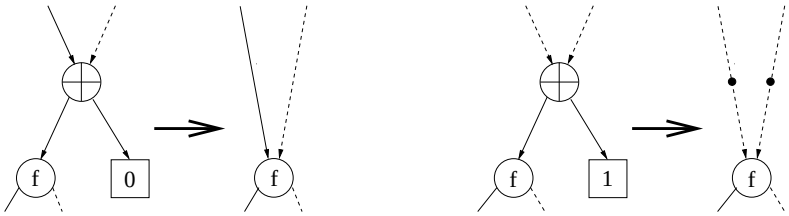


Figure 4.7: Reduction Rule Set for $\oplus$-Nodes Connected to a Terminal Node.

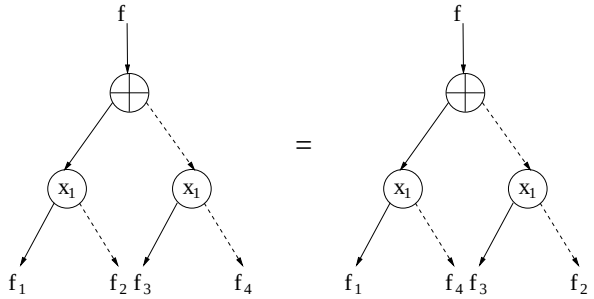


Figure 4.8: Additional Equivalence for $\oplus$-OBDDs.

denoted as f_1, f_2 and f_3, f_4, are disjunct and with the commutative law the following equivalences hold (see also Fig. 4.8):

$$\begin{aligned} f_v &= f_{v_0} \oplus f_{v_1} \\ &= (x_i f_1 + \overline{x_i} f_2) \oplus (x_i f_3 + \overline{x_i} f_4) \\ &= (x_i f_1 \oplus \overline{x_i} f_2) \oplus (x_i f_3 \oplus \overline{x_i} f_4) \\ &= x_i f_1 \oplus \overline{x_i} f_4 \oplus x_i f_3 \oplus \overline{x_i} f_2 \\ &= (x_i f_1 + \overline{x_i} f_4) \oplus (x_i f_3 + \overline{x_i} f_2) \end{aligned}$$

These equivalences are suited to design the following reduction rule: Before any new $\oplus$-node is created, if both of its successors are labeled with the same variable, the equivalence of $f_1 = f_4$ and $f_2 = f_3$ has to be tested and additionally possible reductions have to be carried out. In an extremal case, if $f_1 = f_4$ and at the same time $f_2 = f_3$ holds, the reduction shown in Fig. 4.9 will take place.

Note that this special kind of reduction might also take place, if we consider trees of $\oplus$-nodes, which are connected with branching nodes at their leaves labeled with the same variable

4.4 Equivalence Test

One of the most important reasons responsible for the efficiency of a data structure for the representation of Boolean functions is a fast equivalence test. Starting with the essential task of combinatorial verification, where the equivalence

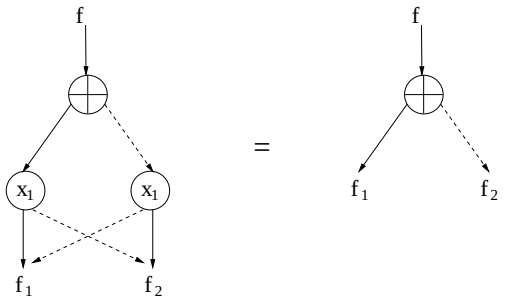


Figure 4.9: Special Case of Equivalence for $\oplus$-OBDDs.

of the representation of a given specification and its implementation has to be tested, at each single step in the process of symbolic simulation we are in need of the equivalence test. The functions represented as OBDDs or $\oplus$-OBDDs have to be uniquely identified, because each time, when a new node is created, it has to be tested, whether the node does already exist or not. Also the whole synthesis procedure is only able to work in an efficient way, if already computed results can be reused, what also necessitates the identification of equivalent nodes. Furthermore, as we have seen in the foregoing chapter, the application of reduction rules also depends on proper identification of equivalences.
For OBDDs, testing the equivalence of two given diagrams, both respecting the same variable order and both being completely reduced is very easy, because they are canonical and thus, uniquely determined. If they are given as two separate data structures their equivalence can be tested by examining, whether the two graphs are isomorphic. If both OBDDs are synthesized within the same process, their top nodes must have the same memory address, if they are representing equivalent functions (strong canonicity) [BRB90] and thus, the equivalence test can be performed by a mere pointer comparison.
For $\oplus$-OBDDs, testing the equivalence of two given diagrams is much more difficult, because this data structure is not canonical anymore. Two given $\oplus$-OBDDs might represent the same Boolean function, although not being isomorphic. E.g. the two $\oplus$-OBDDs might be constructed from different subfunctions that are connected by $\oplus$, and therefore, testing only graph isomorphism is not sufficient for $\oplus$-OBDDs.

4.4.1 Deterministic Equivalence Test

If we are testing graph isomorphism of two given $\oplus$-OBDDs P and Q, representing the Boolean functions f_P and f_Q, both respecting the same variable order π, and the test returns a positive result, then, of course P and Q are representing the same Boolean functions $f_P = f_Q$. But, from a negative test result, it is not possible to conclude that $f_P \neq f_Q$ holds.
Because equivalent $\oplus$-OBDDs may consist out of different subfunctions, all connected by $\oplus$ nodes, equivalence may not in any sense be related to isomorphism. Therefore, an explicit test for equivalence has to check the identity of the function results for every single variable assignment.

Testing the equivalence of two given $\oplus$-OBDDs P and Q, both representing Boolean functions $f_P, f_Q : \{0,1\}^n \to \{0,1\}$, and both respecting the same variable order π, in a naive way requires time $2^{O(n)}$. By checking, whether the identical function value is computed for each given variable assignment $\{a_1, \ldots, a_n\} \in \{0,1\}^n$ for both $\oplus$-OBDDs P and Q, all 2^n variable assignments have to be taken under consideration. For $\oplus$-OBDDs P and Q the function evaluation for a given variable assignment requires time $O(|P| + |Q|)$. Thus, the required time is $2^{O(n)}$.
But, if we admit some slight changes to the model of the $\oplus$-OBDD data structure, we are able to achieve polynomial runtime. Recently, Waack introduced the so called *Parity OBDDs* as a generalized variant of $\oplus$-OBDDs [Waa97].

Definition 4.2 *A Parity OBDD over the variable set $\{x_1, \ldots, x_n\}$ with a given variable ordering π is a directed acyclic graph with one root and sinks labeled with 0 and 1, respectively. Non terminal nodes are labeled with a variable and may have an arbitrary number of outgoing edges labeled with 0 or 1. As for regular OBDDs, the sequence of tests of variables on each path from the root to a sink has to be consistent with π.*

The function f_v computed by a non-terminal node v in a Parity OBDD equals the XOR-synthesis of all its a_i-successors $a_i \in \{0,1\}$ according to the assignment of the variable $x_i = a_i$ the node v is labeled with.

$$f_v = \overline{x_i} \bigoplus_{j=1}^{p} f_{v\overline{x_i}j} + x_i \bigoplus_{k=1}^{q} f_{vx_ik},$$

where p is the number of 0-successors $f_{v\overline{x_i}}$ and q the number of 1-successors f_{vx_i}.
A Parity OBDD according to this definition computes the output 1 for a given variable assignment $\{a_1, \ldots, a_n\} \in \{0,1\}^n$, if and only if the number of paths from the root to the 1-sink w.r.t. the given variable assignment is odd. Because of the arbitrary number of outgoing edges per node, the suitable measure for the size of a Parity OBDD is the number of its edges, which is at most quadratic in the number of nodes.
As in the case of regular $\oplus$-OBDDs, Waack's Parity OBDDs are not a canonical data structure for the representation of Boolean functions. Nevertheless, Waack has shown how a Parity OBDD P representing a Boolean function f_P w.r.t. a given variable order π with a minimal number of nodes can be constructed in polynomial time starting from an arbitrary Parity OBDD for f_P. For the construction of a Parity OBDD with a minimal number of nodes the functions represented by each single node of the Parity OBDD are considered as vectors within a vector space. By an algebraic transformation the task of minimizing the Parity OBDD is transferred to the solution of a set of linear equations.
Furthermore, he has also devised polynomial time algorithms for the operations *synthesis* and *satisfiability* on Parity OBDDs.

Theorem 4.7 *[Waa97] The satisfiability problem for Parity OBDDs is in P.*

Proof: Let P be a Parity OBDD that computes the function f. We want to test, whether $f = 0$.
Let v be the root node of P labeled with the variable x_1. Let $g_1, \ldots, g_k$ be the functions computed at the 0-successors of v and let $g_{k+1}, \ldots, g_{k+l}$ be the functions computed by the 1-successors of v, respectively. Then, f can be expressed with an equation over $GF(2)$ as follows:

$$\begin{aligned} f &= (x_1 + 1)g_1 + \ldots + (x_1 + 1)g_k + x_1 g_{k+1} + \ldots + x_1 g_{k+l} \\ &= g_1 + \ldots + g_k + x_1(g_1 + \ldots + g_{k+l}). \end{aligned}$$

Since, x_1 does not occur in the functions g_i, $1 \leq i \leq k + l$, it holds that $f = 0$ if and only if $g_1 + \ldots + g_k = 0$ and $g_1 + \ldots + g_{k+l} = 0$. But, this requires that the functions g_i are linearly dependent. Some of the functions g_i can be expressed by linear combinations of others and are therefore superfluous. The goal now is to get rid of the superfluous nodes such that the functions computed at the remaining nodes are linearly independent. Then, the given equation must reduce to $f = 0$, if f in fact is the zero-function.
The algorithm works bottom up starting at the sink nodes representing the functions 0 and 1, respectively. Suppose that P is reduced up to some level $l-1$, $1 \leq l \leq n$ and let $b_1, \ldots, b_k$ be the linearly independent functions computed by the nodes at that level. Then, for level l, let there be m nodes computing the functions $f_1, \ldots, f_m$. Each f_i, $1 \leq i \leq m$ can be expressed analogously to f in the above equation in terms of the $2k$ base functions $b_1, \ldots, b_k, x_l b_1, \ldots, x_l b_k$. In this way we get a set of m linear equations:

$$\begin{pmatrix} a_{1,1} & \ldots & a_{1,2k} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \ldots & a_{m,2k} \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ \vdots \\ b_k \\ x_l b_1 \\ \vdots \\ x_l b_k \end{pmatrix} = \begin{pmatrix} f_1 \\ \vdots \\ f_m \end{pmatrix},$$

where $a_{i,j} \in \{0, 1\}$ are the corresponding coefficients of each f_i.
Now, with Gaussian elimination the matrix is transformed into a lower triangular form and simultaneously the same transformations are computed at the vector $(f_1, \ldots, f_m)^T$. Suppose, the resulting matrix has $d \leq m$ 0-rows at the bottom. Then, there are d equations such that $f_{m-d+1}, \ldots, f_m$ can be expressed in terms of $f_1, \ldots, f_{m-d}$ and the nodes computing $f_{m-d+1}, \ldots, f_m$ become superfluous. These nodes are eliminated and all incoming edges are redirected according to the equations obtained for $f_{m-d+1}, \ldots, f_m$ in order to get an equivalent Parity OBDD. Now, all nodes up to level l compute linearly independent functions. Since, Gaussian elimination is required for each level of P, the running time sums up to $O(n|P|^3)$, where $|P|$ denotes the number of nodes of P. □

Now, by computing, whether for Parity OBDDs P, Q the expression $f_P \oplus f_Q$ is satisfiable, a deterministic equivalence test can be performed in polynomial

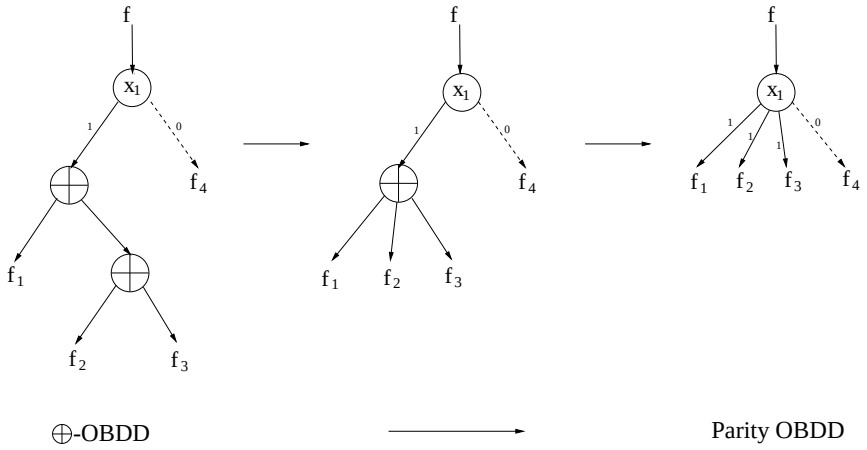


Figure 4.10: Transformation from ⊕-OBDDs to Parity OBDDs.

time.

The ⊕-OBDD model can easily be transformed into the alternative Parity OBDD data structure.

Theorem 4.8 *Let P be a ⊕-OBDD that is representing an arbitrary Boolean function $f_P : \{0,1\}^n \to \{0,1\}$. Then f_P can be represented by a Parity-OBDD Q of size $|Q| = O(|P^2|)$.*

Proof: Note that the size of a ⊕-OBDD is the number of nodes, and for Parity OBDDs the size equals the number of edges. The sink nodes and every regular branching node (P_B) in a ⊕-OBDD P can be directly regarded as a node of a Parity OBDD Q. For every ⊕-node v ($P_\oplus$) in P we are introducing up to two edges into Q, connecting every predecessor of the ⊕-node with v_0 and v_1. Thus, the size of Q is bounded by $|Q| \leq 2|P_B| + 2|P_B| \cdot |P_\oplus| = O(|P|^2)$. □

For transforming a ⊕-OBDD into a Parity OBDD we are proceeding in the following way: First, we sum up all ⊕-nodes that have a direct connection to each other, resulting in meta-⊕-nodes (see Chapter 5) with an arbitrary number of successors. Then, every branching node v labeled with the variable x_i and connected to a ⊕-node being one of its successors v_0 or v_1 is directly connected to all the successors of v_0 and v_1, while edges leading to successors of v_0 are labeled with 0, and edges leading to successors of v_1 are labeled with 1, respectively (see Fig. 4.10).
For a transformation of a Parity OBDD into a ⊕-OBDD, we simply reverse the whole process: Let v be a node of a Parity OBDD, having n_1 1-successors $v_{1_1}, \ldots, v_{1_{n1}}$ and n_0 0-successors $v_{0_1}, \ldots, v_{0_{n0}}$. The node v is transformed into a regular binary branching node by connecting it to two new meta-⊕-nodes v_0 and v_1. v_0 is connected to all n_0 0-successors of the former Parity OBDD node v, and v_1 with all n_1 1-successors, respectively. In the last step, v_0 and v_1 are transformed into trees of $\lceil \log(n_0) \rceil$ and $\lceil \log(n_1) \rceil$ regular binary ⊕-nodes, with $v_{0_1}, \ldots, v_{0_{n0}}$ and $v_{1_1}, \ldots, v_{1_{n1}}$ as their leaf nodes.

Based on the model of Parity OBDDs the fastest known deterministic equivalence test requires time cubic in the number of nodes. But, in symbolic simulation for the creation of a single node several equivalence tests have to be performed: First, we have to verify, whether the operation that is computing a new node has already been carried out before. Then, before the node can be created we have to test, whether a rule out of the set of reduction rules can be applied. Finally, we have to verify, if the node to be created does already exist. For all these operations equivalence tests have to be performed. Circuit designs in a practical working environment often comprise millions of nodes and thus, the deterministic equivalence test although having polynomial runtime related to its number of inputs is much too slow and not suitable for working in a commercial environment.

4.4.2 Probabilistic Equivalence Test for Boolean Functions

For achieving an equivalence test for $\oplus$-OBDDs that fulfills the necessary performance requirements, we have to fall back on a probabilistic approach. In general the method that is described here is based on the arithmetic transformation of the Boolean function represented as a $\oplus$-OBDD into a multi-variate and multi-linear polynomial over a finite domain. Then, the equivalence of two $\oplus$-OBDDs transformed into polynomials can be easily tested by applying a standard randomized procedure. This technique has already been successfully applied to OBDDs or FBDDs, and can be extended to $\oplus$-OBDDs, too.
The algorithm itself belongs to the complexity class ***co-RP***, which means that the algorithm has a one-sided error probability. But, given that the error probability is $\frac{1}{2}$, the k-fold repetition of the test leads to a reduction of the error probability to $\frac{1}{2^k}$. Blum, Chandra, and Wegman described a probabilistic equivalence test for FBDDs [BCW80] based on a probabilistic zero-test for polynomials developed by Schwartz[Sch80] and Zippel[Zip70]. This concept was extended and generalized by Jain, Bittner, Fussel, and Abrahams to be applied for OBDDs [JBF+92]. To develop a probabilistic equivalence test for $\oplus$-OBDDs, we take a closer look on this method.

Probabilistic Equivalence Test for OBDDs

For every node v in an OBDD, a *Hash Code* a_{f_v} for the Boolean function $f_v(x_1, \ldots, x_n)$ represented by v is created. To achieve this, for each input variable $x_1, \ldots, x_n$ a random integer element $a_i \in \mathbb{Z}_p$, p prime - the finite field of integers *modulo* p is chosen. Then, a functionally transformed version of f_v, denoted as $A[f_v]$ (*A-transformation of* f_n) that is represented by a field polynomial over $\mathbb{Z}_p$ is generated and evaluated with the chosen random elements a_i.
We are now defining the A-transformation for a Boolean Function: Let $b_i \in \{0, 1\}$ represent the truth value assigned to the variable x_i. For representing the value assignment to a Boolean variable $x_i = b_i$ we introduce the expression $w(b_i, x_i)$ that can be expanded for the representation of a whole minterm to $w(b_1, \ldots, b_n, x_1, \ldots, x_n)$.

Boolean operation $\otimes$	0,1-equivalent extended operation $\otimes_{\mathbb{Z}_p}$
$\neg f$	$\neg_{\mathbb{Z}_p} f = 1 - f$
$f_1 \wedge f_2$	$f_1 \wedge_{\mathbb{Z}_p} f_2 = f_1 \cdot f_2$
$f_1 \vee f_2$	$f_1 \vee_{\mathbb{Z}_p} f_2 = f_1 + f_2 - f_1 \cdot f_2$
$f_1 \oplus f_2$	$f_1 \oplus_{\mathbb{Z}_p} f_2 = f_1 + f_2 - 2 \cdot f_1 \cdot f_2$

Table 4.2: Extended Boolean Operations

Definition 4.3 *Let* $w : \mathbb{Z}_p \times \mathbb{Z}_p \to \mathbb{Z}_p$ *be defined as* $w(b,x) = bx + (1-b)(1-x)$. *Then, define* $w : \mathbb{Z}_p^{2n} \to \mathbb{Z}_p$ *as*

$$w_n(b_1, \ldots, b_n, x_1, \ldots, x_n) = \prod_{i=1}^{n} w(b_i, x_i)$$

Now, we can sum up all minterm representations w_n to represent the A-transform of a given Boolean function f. For readability we abbreviate $b_1, \ldots, b_n$ with b and $x_1, \ldots, x_n$ with x.

Definition 4.4 *Given a function* $f : \{0,1\}^n \to \mathbb{Z}_p$, *the polynomial* $A[f] : \mathbb{Z}_p^n \to \mathbb{Z}_p$ *is defined as*

$$A[f](x) = \sum_{b \in \{0,1\}^n} f(b) \cdot w_n(b,x).$$

If two functions f_1 and f_2 compute the same result for any Boolean vector $b \in \{0,1\}^n$, they are called *0,1-equivalent*. Applying the A-transform to a Boolean function increases the domain from $\{0,1\}$ to $\mathbb{Z}_p$, but $A[f]$ still yields the same values as f, when evaluated on a Boolean vector $(a_1, \ldots, a_n) \in \{0,1\}^n$.

Theorem 4.9 *The functions* f *and* $A[f]$ *are 0,1-equivalent,* $f \overset{0,1}{=} A[f]$.

Note that 0,1-equivalent functions on the domain $\mathbb{Z}_p$ do not have to be equal, e.g. $f = x^k$, $k \in \mathbb{N}$. Here, $x^k \overset{0,1}{=} x$, but $x^k \neq x$, if x^k is defined over the domain $\mathbb{Z}_p$. For computing the A-transform of a more complex Boolean expression Φ we are replacing simple Boolean subfunctions Φ_1 by their corresponding 0,1-equivalent field function $A[\Phi_1]$ (shown in the 2nd column of Table 4.2) incrementally until we finally get $A[\Phi]$.
If an arbitrary extended Boolean operation $\otimes_{\mathbb{Z}_p}$ is to be applied to two functions f_1 and f_2, which are defined on a disjoint variable set, then $A[f_1 \otimes_{\mathbb{Z}_p} f_2] = A[f_1] \otimes_{\mathbb{Z}_p} A[f_2]$ holds. Otherwise, if the two variable sets are not disjoint, the following theorem holds.

Theorem 4.10 *For all extended Boolean operations* $\otimes_{\mathbb{Z}_p}$ *it holds that*

$$A[f_1 \otimes_{\mathbb{Z}_p} f_2] = A[A[f_1] \otimes_{\mathbb{Z}_p} A[f_2]].$$

This means that every Boolean operation can be extended in a way that no exponents $k > 1$ will be created. Since $x \otimes_p y$ must be a polynomial in x and y we have the 0,1-equivalence

$$x \otimes_p y \overset{0,1}{=} c_1 + c_2 \cdot x + c_3 \cdot y + c_4 \cdot x \cdot y$$

for some $c_1, c_2, c_3, c_4 \in \mathbb{Z}_p$ and any extended Boolean operation can be assumed to have this bilinear form.
To achieve identical polynomials $A[f_1] = A[f_2]$ for functions f_1 and f_2, which are equivalent on the Boolean domain $f_1 \overset{0,1}{=} f_2$, it is sufficient to extend the Boolean operations to operations over a finite domain in that way that higher exponents cannot be created and to exclude higher exponents in general. Thus, we can define the following computation rules for obtaining the polynomial $A[f]$:

Theorem 4.11 *For any field function $f_1, f_2 : \mathbb{Z}_p \times \mathbb{Z}_p \to \mathbb{Z}_p$ and any constant $c \in \mathbb{Z}_p$ the following holds:*

(1) $A[c \cdot f_1] = c \cdot A[f_1]$

(2) $A[f_1 \pm f_2] = A[f_1] \pm A[f_2]$

(3) $A[f_1 \cdot f_2] = A[f_1] \cdot A[f_2]$, *if f_1 and f_2 are disjoint*

(4) $A[c] = c$

(5) $A[x_i] = x_i$.

(6) $A[f_1] = (1 - x) \cdot A[f_{1\overline{x}}] + x \cdot A[f_{1x}]$

Proof: (1), (2), (4), and (5) follow directly from the definition of the A-transform. (3) is the application of the rule concerning the application of operations on functions with disjoint variable sets. (6) follows from the definition of $A[f_1]$ after reordering the addends. □

With these rules, the A-transform can be applied to any arbitrary arithmetic expression, but the explicit evaluation of the polynomial often would be too difficult and thus, would also require to much time. For a probabilistic equivalence check, it is sufficient to compare the A-transforms of two given functions only in distinct instances. But, care must be taken, when substituting variables with numeric values, if we want to maintain a proper hash code. This means that we can only substitute a variable by a constant in an expression $A_1 \otimes_p A_2$, if only one of the two expressions A_1 and A_2 is dependent of this variable. From the given rules for the A-transform and an arbitrary extended Boolean operation we have

$$A[f_1 \otimes_p f_2] = c_1 + c_2 A[f_1] + c_3 A[f_2] + c_4 A[f_1 \cdot f_2]$$

The term $A[f_1 \cdot f_2]$ prevents the direct combination of the hash codes for f_1 and f_2 and it can be eliminated for the conditions $A[f_1 \cdot f_2] = 0$, which requires

f_1 and f_2 to be *orthogonal*, or, if for f_1 and f_2 the dual of the orthogonality condition holds, then $A[f_1] + A[f_2] + A[f_1 \cdot f_2] = 1$.

Based on this theoretical foundation we can start a general analysis of the error probability.

Theorem 4.12 *For any polynomial f on $\mathbb{Z}_p$ over n variables such that $f \neq 0$, and any $S \subseteq \mathbb{Z}_p$, $s = |S|$, there are at least $(s-1)^n$ vectors $v \in S^n$ such that $A[f](v) \neq 0$.*

Proof: The proposition can be shown by induction on n. For $n = 0$ the polynomial is a constant $c \neq 0$. Otherwise, let $x_i \in \{x_1, \ldots, x_n\}$ be a variable of f. Then we can expand $A[f]$ in the following way:

$$\begin{aligned} A[f] &= (1 - x_i) \cdot A[f_{\overline{x_i}}] + x_i \cdot A[f_{x_i}] \\ &= x_i \cdot (A[f_{x_i}] - A[f_{\overline{x_i}}]) + A[f_{\overline{x_i}}] \\ &= x_i \cdot A[f_{x_i} - f_{\overline{x_i}}] + A[f_{\overline{x_i}}] \end{aligned}$$

If $f_{\overline{x_i}} = f_{x_i}$, then $A[f_{x_i} - f_{\overline{x_i}}] = 0$. By induction, there are at least $(s-1)^{n-1}$ vectors $v \in S^{n-1}$ such that $A[f_{\overline{x_i}}] \neq 0$. x_i can be chosen arbitrarily and thus, we have $s(s-1)^{n-1} \geq (s-1)^n$ possible assignments, which compute $A[f] \neq 0$.

Otherwise, if $A[f_{\overline{x_i}}] = 0$, then, there are at least $(s-1)^{n-1}$ vectors $v \in S^{n-1}$ such that $A[f_{x_i} - f_{\overline{x_i}}] \neq 0$. Since, x_i can be chosen arbitrarily with

$$x_i \neq -\frac{A[f_{\overline{x_i}}]}{A[f_{x_i} - f_{\overline{x_i}}]},$$

each provides $(s-1)$ additional vectors of length n and therefore, we have $(s-1)(s-1)^{n-1} = (s-1)^n$ possible assignments, which compute $A[f] \neq 0$. □

Now, we can easily compute the probability that the A-transform of two arbitrarily chosen field functions f_0, f_1 differs on a randomly chosen input $v \in \mathbb{Z}_p^n$, if $A[f_0]$ and $A[f_1]$ are different.

Theorem 4.13 *Let f_0 and f_1 be any two field functions of n variables, such that $A[f_0] \neq A[f_1]$, and let v be a vector of length n, whose elements are randomly chosen from $\mathbb{Z}_p$. Then*

$$Prob(A[f_0](v) \neq A[f_1](v)) \geq \left(\frac{p-1}{p}\right)^n$$

Proof: For any $v \in \mathbb{Z}_p^n$, $A[f_0](v) \neq A[f_1](v)$ holds, if and only if $A[f_0 - f_1](v) \neq 0$. We have just shown that there are at least $(p-1)^n$ vectors out of possible p^n that are full filling this proposition. □

The error probability is therefore bounded by $\epsilon = 1 - (1 - \frac{1}{p})^n \approx 1 - e^{-n/p}$ and if $p \gg n$, then $1 - e^{-n/p} \approx \frac{n}{p}$. Thus, the error probability can be reduced

by either choosing a larger field $\mathbb{Z}_p$, or by independently performing several random tests.
The Boolean functions under consideration are hashed on the p elements of the field. With ideal hashing the probability of a collision would be $\frac{1}{p}$, thus making $\frac{n}{p}$ a good choice for a simple hash function as the one we have designed.
In the following section, this technique will be adapted to $\oplus$-OBDDs, as it was first proposed by Gergov and Meinel [GM93].

4.4.3 Applying the Probabilistic Equivalence Test to $\oplus$-OBDDs

For applying the probabilistic equivalence test to $\oplus$-OBDDs we use polynomials over a Galois field $GF(2^m)$, $m \in \mathbb{N}$. $GF(2^m)$ is of characteristic $char(GF(2^m)) = 2$, which simplifies the computation rules for creating polynomials that are generated from $\oplus$-OBDDs representing Boolean functions. For improving the readability we write p_f for the polynomial that is described by the A-transform $A[f]$ of a Boolean function f.

Definition 4.5 *A polynomial p_f in $GF(2^m)$ representing the A-transform $A[f]$ of a Boolean function f, is defined in the following way:*

$$p_f = \sum_{a \in f^{-1}(1)} \prod_{i=1}^{n} x_i^{a_i}(1 - x_i)^{(1-a_i)}$$

where $a = (a_1, \ldots, a_n) \in \{0,1\}^n$.

The following properties can be shown easily:

Lemma 4.1 *Let $f, g : \{0,1\}^n \to \{0,1\}$ be Boolean functions and $GF(2^m)$, $m \in \mathbb{N}$ a Galois field.*

(1) The polynomials $p_f, p_g \in GF(2^m)$ are linear in all of their variables $x_i \in \{x_1, \ldots, x_n\}$, and $f = g$ if and only if $p_f = p_g$.

(2) If $f^{-1}(1)$ and $g^{-1}(1)$ are disjoint then $p_{f \vee g} = p_{f \oplus g} = pf + p_g$.

(3) If there does not exist a variable $x_i \in \{x_1, \ldots, x_n\}$, such that both functions f and g are depending on x_i, then $p_{f \wedge g} = p_f \cdot p_g$.

(4) $p_{f \oplus g} = p_f + p_g$.

Proof: (1)-(3) follow directly from the theorems of the last section
(4) $p_{f \oplus g} = p_f + p_g - 2 \cdot p_f \cdot p_g$. With $char(GF(2^m)) = 2$ the subtract multiplied by the factor 2 can be reduced to 0. □

If we now consider the elements of $GF(2^m)$ as bit vectors of length m, then the addition can be performed by bitwise application of EXOR. Now, a polynomial p_v for a node v of a $\oplus$-OBDD P_{f_v} representing the Boolean function f_v can easily be computed.

Definition 4.6 *Let P_f be a $\oplus$-OBDD that is representing the Boolean function $f : \{0,1\}^n \to \{0,1\}$. Let v_0, v_1 be the successor nodes of node v. We associate each node v of P_f with a polynomial $p_v : (GF(2^m))^n \to GF(2^m)$:*

$$p_v = \begin{cases} 0(1) & \text{if } v \text{ is a 0-sink(1-sink)}, \\ p_{x_i} \cdot p_{v1} + (1 - p_{x_i}) \cdot p_{v0} & \text{if } v \text{ is labeled with } x_i, \\ p_{v0} + p_{v1} & \text{if } v \text{ is a } \oplus\text{-node}. \end{cases}$$

The polynomial p_f for the Boolean function f represented by the $\oplus$-OBDD P_f is the polynomial computed for the root node of P_f. Note that for the Boole/Shannon expansion of a branching node v by the variable x_i, the addends $x_i \cdot f_{x_i}$ and $\overline{x_i} \cdot f_{\overline{x_i}}$ are disjoint conjunctions, and that the sum $x_i \cdot f_{x_i} + \overline{x_i} \cdot f_{\overline{x_i}}$ is also disjunctive.
For reasons of efficiency the polynomials of the $\oplus$-OBDDs to be compared will not be computed explicitely, but they will be compared at certain randomly chosen instances as shown in the previous section. Therefore, for every variable $x_i \in \{x_1, \ldots, x_n\}$ of the Boolean function $f : \{0,1\}^n \to \{0,1\}$ represented by the $\oplus$-OBDD P_f we choose the elements $A[x_i] = p_{x_i} \in GF(2^m)$, $i = (1, \ldots, n)$ independently and uniformly at random. With these values the hash codes $p_v(p_{x_1}, \ldots, p_{x_n})$ identifying a unique node v of the $\oplus$-OBDD P_f can be computed efficiently. Note that the hash codes for each node of a $\oplus$-OBDD are computed, when the node itself is created during $\oplus$-OBDD synthesis. Thus, the computation of the hash code depends only on the already computed hash codes of its successors (for more details see *Implementation of the Probabilistic Equivalence Test*). This hash code for the Boolean function f is also often referred as *Boolean signature* or simply *signature* sig_f.
Now, everything is prepared for the definition of an algorithm for the probabilistic equivalence test for $\oplus$-OBDDs. Let P_f and P_g be two $\oplus$-OBDDs representing the Boolean functions $f, g : \{0,1\}^n \to \{0,1\}$. Let $m \in \mathbb{N}$ such that $m > log(n) + 1$, and, thus, $|GF(2^m)| > 2n$. Assume that $a_1, \ldots, a_n \in GF(2^m)$ are generated independently and uniformly at random.

Theorem 4.14 *For the Boolean signatures p_f and p_g computed for the $\oplus$-OBDDs P_f and P_g it holds that*

(1) $p_f(a_1, \ldots, a_n) = p_g(a_1, \ldots, a_n)$, if $f = g$, and

(2) $Prob(p_f(a_1, \ldots, a_n) = p_g(a_1, \ldots, a_n)) < \frac{1}{2}$, if $f \neq g$.

Proof: (1) follows directly from the definition of the A-transform $A[f]$ of a Boolean function f. (2) With theorem 3.13 it follows that

$$Prob(p_f(a_1, \ldots, a_n) = p_g(a_1, \ldots, a_n)) < 1 - (\frac{p-1}{p})^n \quad \text{for } f \neq g$$

with p denoting the size of $|GF(2^m)| > 2n$. The error bound can be estimated in the following way:

$$1 - (\frac{p-1}{p})^n < 1 - (\frac{2n-1}{2n})^n = 1 - (1 - \frac{1}{2n})^n < 1 - e^{-\frac{1}{2}} < \frac{1}{2}.$$

Input: $\oplus$-OBDDs P_f, P_g.
Output: If $f = g$ the algorithm answers *yes*, otherwise it returns *no* with probability greater than $\frac{1}{2}$.

```
equivalence(P_f, P_g) {
   choose independently and uniformly at random a_1,...,a_n;
   compute p_f(a_1,...,a_n) and p_g(a_1,...,a_n);
   if ( p_f(a_1,...,a_n)==p_g(a_1,...,a_n) ) {
     return(yes);
   } else {
     return(no);
   }
}
```

Figure 4.11: Algorithm for a Probabilistic Equivalence Test for $\oplus$-OBDDs.

□

See Fig 4.11 for the algorithm in pseudocode.
Since SAT and EQU are related in the way that $SAT(f)$ can be computed by testing $EQU(f, 0)$ and negating the achieved result, we conclude also the following theorem:

Theorem 4.15

(1) $SAT_{\oplus-OBDD}$ *is probabilistic feasible,* $SAT_{\oplus-OBDD} \in \mathbf{R}$.

(2) $EQU_{\oplus-OBDD}$ *is probabilistic feasible,* $EQU_{\oplus-OBDD} \in$ **co-R**.

4.4.4 Determining the Error Probability

Based on the theorems 4.13 and 4.14, now, a closer look on the reliability of this hashing technique is taken. According to the given definitions, the Boolean signatures $p_f(a_1, \ldots, a_n)$ and $p_g(a_1, \ldots, a_n)$ of two given Boolean functions f and g, for $f = g$ are certainly equal, but, on the other hand if two given signatures $p_f(a_1, \ldots, a_n)$ and $p_g(a_1, \ldots, a_n)$ are equal, then the functions f and g are equal only with a certain probability.

Definition 4.7 *A pairwise degeneracy between unequal Boolean functions* $f, g : \{0,1\}^n \to \{0,1\}$, $f \neq g$, *occurs for those assignments* $(a_1, \ldots, a_n)$ *of input signatures that cause* $p_f(a_1, \ldots, a_n) = p_g(a_1, \ldots, a_n)$. *A degeneracy in a* $\oplus$*-OBDD occurs for those assignments, where a pairwise degeneracy exists between any pair of vertices in the* $\oplus$*-OBDD.*

For $\oplus$-OBDDs we can adopt the bound for pairwise degeneracy given by Brace [Bra92].

Theorem 4.16 *On a $\oplus$-OBDD P containing $|P|$ nodes, the proportion of signature assignment to variables, which will cause a degeneracy somewhere in the $\oplus$-OBDD is less than $\frac{|P|^2}{2 \cdot |GF(2^m|)}$.*

Proof: There are $\binom{|P|}{2}$ pairs of nodes in the $\oplus$-OBDD P. Let $r, s \in \mathbb{N}$, $1 \leq r, s \leq n$. If we consider nodes v_r of a distinct level $l(v_r) = x_r$ in P, all labeled with the variable x_r (or, if v_r is a $\oplus$-node and its highest successor node is labeled with x_r. These nodes are actually stored in the same hash table (*unique table*) as the variable x_r, see section *Implementational Details of the $\oplus$-OBDD Package*), then a pair of nodes (v_p, v_q) is of level r, if $max(l(v_q), l(v_p)) = x_r$. Note that the nodes labeled with the bottom variable next to the sink are level 1 and nodes labeled with the top variable are of the highest level. A level r degeneracy is called the degeneracy between a level r pair such that no degeneracies occur between any level k pairs for any $k < r$. For determining the maximum number of possible level r degeneracies, we require the following lemma:

Lemma 4.2 *For each level r pair (v_q, v_p) in an $\oplus$-OBDD P there are at most $|GF(2^m)|^{r-1}$ signature assignments to the variables of level k, $k \leq r$ which will lead to a level r degeneracy between v_q and v_p.*

Proof: By induction on r. For $r = 1$ at level 1, there are only the $\oplus$-OBDDs representing the constant functions $f(x_1) = 1/0$, or $f(x_1) = x_1/\overline{x_1}$. For every possible pair, the lemma allows one degenerate signature assignment. For $(x_1, 1)$ / $(\overline{x_1}, 0)$ the only degenerate assignment is $sig_{x_1} = 1$, for $(x_1, 0)$ / $(\overline{x_1}, 1)$ the only degenerate assignment is $sig_{x_1} = 0$, respectively. For $(x_1, \overline{x_1})$ the only degenerate assignment is the unique value which is its own additive inverse.
By induction hypothesis the lemma holds for $\oplus$-OBDDs having $r-1$ levels. Of the $|GF(2^m)|^{r-1}$ possible signature assignments to the $r-1$ previous variables, some of them might lead to an earlier degeneracy, but they are not considered for this proof, because a level r degeneracy only occurs, when there are no degeneracies in lower levels. Thus, there are at most $|GF(2^m)|^{r-1}$ assignments through level $r-1$ which could possibly allow a level r degeneracy. Blum, Chandra, and Wegman [BCW80] showed that there is at most one assignment of a signature to x_r such that $sig_{v_q} = sig_{v_p}$ for each of these $|GF(2^m)|^{r-1}$ cases: A pairwise degeneracy for (v_q, v_p) exists, if $sig_{v_q} = sig_{v_p}$. We can decompose v_q and v_p with Boole/Shannon decomposition by variable x_r:

$$sig_{x_r} \cdot sig_{v_{q_{x_r}}} + sig_{\overline{x_r}} \cdot sig_{v_{q_{\overline{x_r}}}} \quad = \quad sig_{x_r} \cdot sig_{v_{p_{x_r}}} + sig_{\overline{x_r}} \cdot sig_{v_{p_{\overline{x_r}}}}$$

Now, either $v_{q_{x_r}} \neq v_{p_{x_r}}$, or $v_{q_{\overline{x_r}}} \neq v_{p_{\overline{x_r}}}$. W.l.o.g. assume $v_{q_{x_r}} \neq v_{p_{x_r}}$ (the other case is symmetric) and let $(x_1, \ldots, x_{r-1})$ be assigned values such that $sig(v_{q_{x_r}}) \neq sig(v_{p_{x_r}})$. The above equation reduces to

$$\begin{aligned} sig_{x_r} \cdot \left(sig_{v_{q_{x_r}}} - sig_{v_{p_{x_r}}} - sig_{v_{q_{\overline{x_r}}}} + sig_{v_{p_{\overline{x_r}}}} \right) &= sig_{x_r} \cdot \left(sig_{v_{q_{x_r}}} - sig_{v_{p_{x_r}}} \right) \\ &= 0 \end{aligned}$$

.

Since only $sig_{v_{q_{x_r}}} \neq sig_{v_{p_{x_r}}}$ is considered and all computations are performed in a finite field (i.e. each signature has a unique multiplicative inverse except for the unique signature 0) there is at most one value for sig_{x_r} that solves the equation. Since there are $r-1$ other variables, there are at most $|GF(2^m)|^{r-1}$ assignments to the r variables which could cause a level r degeneracy between v_q and v_p. □

Thus, the portion of possible candidate assignments, where a pairwise degeneracy might occur is at most $\frac{|GF(2^m)|^{(r-1)}}{|GF(2^m)|^r} = \frac{1}{|GF(2^m)|}$. Now, a degeneracy might occur in at most $\frac{1}{|GF(2^m)|}$ of all signature assignments and the chance of finding no degeneracy among all possible pairs of nodes is greater than

$$1 - \binom{|P|}{2} \cdot \frac{1}{|GF(2^m)|} > 1 - \frac{|P|^2}{2} \cdot \frac{1}{|GF(2^m)|} = 1 - \frac{|P|^2}{2 \cdot |GF(2^m)|}$$

□

Note that the probability of pairwise degeneracy has not been bounded to $\frac{1}{|GF(2^m)|}$ throughout all pairs and all variable assignments in the $\oplus$-OBDD. A degeneracy in the lower levels of the $\oplus$-OBDD does increase the chance of degeneracy in some higher levels. The proof is only concerned for the portion of variable assignments, which yield a $\oplus$-OBDD completely free of degeneracies, not the proportion of pairwise degeneracies over all of those assignments.
The reliability of Boolean signatures can be amplified by providing a larger test domain, i.e. increasing m of $|GF(2^m)|$, or by performing multiple independent tests, i.e. providing several parallel signatures.

Theorem 4.17 *Let P be a $\oplus$-OBDD. Using s parallel signatures, the proportion of signature assignments that lead to a pairwise degeneracy between any pair among $|P|$ functions is at most $\frac{|P|^2 \cdot n^s}{2 \cdot |GF(2^m)|^s}$.*

Proof: By theorem 4.13 and 4.14 we have already shown that a particular pairwise degeneracy occurs in at most $\frac{n}{|GF(2^m)|}$ of the possible initial signature assignments. By using s parallel signatures the risk of pairwise degeneracy comes to $\left(\frac{n}{|GF(2^m)|}\right)^s$. With $|P|$ nodes, there are less than $\binom{|P|}{2}$ possible pairs of nodes, therefore the chance of having at least one pairwise degeneracy among them is less than $\frac{|P|^2 \cdot n^s}{2 \cdot |GF(2^m)|^s}$. □

Compared to theorem 4.16 this bound is not as tight for $s = 1$. But, it is safe even for assuming the worst pathological interaction between all functions that are represented.

4.4.5 Implementation of the Probabilistic Equivalence Test

The implementation of the probabilistic equivalence test has been an essential part in the development of the $\oplus$-OBDD package and the main focus of that implementation was concerned on questions of efficiency. The development

of the $\oplus$-OBDD package started in a 32-bit UNIX environment. Thus, the probabilistic equivalence test is based on word level 32-bit arithmetic, but it can also easily be expanded for 64-bit arithmetic.
We consider the use of 32-bit words for representing a hash code of a polynomial that is representing a Boolean function, i.e. we consider the field $GF(2^{31})$ as a basis. We can work in $GF(2^m)$ as in $\mathbb{F}_2[x]/p(x)$ with $p(x)$ denoting an irreducible polynomial of degree m. Such polynomials can be found easily, e.g. $x^{2 \cdot 3^k} + x^{3^k} + 1$ is irreducible in $\mathbf{F}_2[x]$ for each $k \in \mathbb{N}$ [LN86]. For taking advantage of the full 31-bit length we are choosing the irreducible polynomial $x^{31} + x^3 + 1$, which computes a bit vector of the value $2^{31} + 2^3 + 1$. All computations in $\mathbf{F}_2[x]$ have to be computed modulo the chosen irreducible polynomial. The 32nd bit is used as a temporary storage for the carry of the multiplication of two 31-bit numbers. Thus,we are avoiding an overflow out of the given word bounds. While addition and subtraction can be computed by a bitwise XOR operation on the bit vectors, the multiplication algorithm for two r-bit numbers using only a $r+1$ bit variable for intermediate results is working in the following way:

Let $(a_{r-1} \dots a_0)$ and $(b_{r-1} \dots b_0)$ be bit vectors of length r, and let $(i_{r-1}, \dots, i_0)$ be the irreducible polynomial for $\mathbf{F}_2[x]$. Now, for every bit b_i of $(b_{r-1} \dots b_0)$ we add $b_i \cdot 2^i \cdot (a_{r-1} \dots a_0)$ mod $(i_{r-1}, \dots, i_0)$ to the intermediate result $(c_{r-1} \dots c_0)$. Note that multiplication by 2^i is equivalent to shifting the bit vector to the left by i digits. For avoiding an overflow the multiplication is split in single shift left and addition operations for each digit. Thus, the multiplication operation in $GF(2^r)$ requires $O(r)$ elementary operations [STP86].
As mentioned in the previous chapter, the computation of the hash code for a branching node v labeled with the variable $x_i \in \{x_1, \dots, x_n\}$ in a $\oplus$-OBDD P depends only on the already computed hash codes of its two successors v_0 and v_1 and the hash code for the variable x_i. Thus, we are computing

$$sig_v = sig_{x_i} \cdot sig_{v_1} + (1 - sig_{x_i}) \cdot sig_{v_0}.$$

We require two multiplications and one addition for computing the hash code of a branching node and only one addition for computing the hash code of a $\oplus$-node.
For storing the Boolean signature of a $\oplus$-OBDD node we have two different possibilities: First, if our goal is to use as little memory as possible, we dont reserve additional space in each node for storing its signature, but we have to recompute the signature every time it is required. Note that this approach necessitates also the recomputation of the signatures of all the node's predecessors. Thus, we are saving space, but the larger the $\oplus$-OBDD gets, the longer the computation of a signature in the upper levels of the $\oplus$-OBDD will take, although its time complexity is linearly bound by $O(|P|)$. But, equivalence testing is one of the most frequent operations in $\oplus$-OBDD synthesis and manipulation. Therefore, we decided to spend additional memory for each node for storing one or more signatures. For each independent signature additional 32 bit per node have to be invested.

s	Prob. of Degeneracy Φ_d
1	-
2	$1.08 \cdot 10^{-1}$
3	$5.0 \cdot 10^{-9}$
4	$2.35 \cdot 10^{-16}$

Table 4.3: Probability of Degeneracy using Theorem 3.16 for s 32-bit Signatures

$$\left(1 - \left(1 - \frac{1}{|GF(2^m)|}\right)^n\right)^1 \quad \text{vs.} \quad \left(1 - \left(1 - \frac{1}{2}\right)^n\right)^m$$

(a) (b)

Figure 4.12: Comparing the Error Probability of Signatures(a) and Simulation(b)

For determining, what is a sufficient number of signatures to operate with in our package, we computed the error probabilities of degeneracy Φ_d for an arbitrary $\oplus$-OBDD of very large dimensions: We considered a $\oplus$-OBDD P with $|P| = 10^7$, defined over $n = 100$ variables, within the given finite Field $GF(2^m)$, $m = 31$. s denotes the number of independent signatures (see Table 4.3).

For all performed experiments $s = 3$ signatures were proven to be completely sufficient to prevent any degeneracy (See chapter *Experimental Setup for Testing the Reliability of the Probabilistic Equivalence Test*). Note that the given error bound is a rather conservative one. In fact it is never the case that every pair of $\oplus$-nodes displays the worst possible pathological interaction with respect to the probability of pairwise degeneracy.
If we compare Boolean signature based verification with random Boolean simulation, where distinct random bit vectors are serving as inputs and the achieved outputs of the two designs to be verified are compared, we might get a better insight into the achieved error bound. Using signatures for each input variable $x_i \in \{x_1, \ldots, x_m\}$ a $log(|GF(2^m)|) = m$ random bit vector is assigned to and thus, we are using $m \cdot n$ bits. The random Boolean simulation should use the same number of bits to be tested and thus, it performs m runs with random bit vectors of length n. In the worst case, the two functions that should be compared, differ exactly in one minterm and the probability not to select it as one of the m input patterns is $(1 - \frac{1}{2^n})^m$. In fact we have to compare the probabilities given in Fig. 4.12:

Because $|GF(2^m)| >> n$ and $|GF(2^m)| >> 2$, under the same premises the signature based approach results in a much higher reliability. For a better understanding of this argument we can use *assignment tables*, an analogon for a truth table for an algebraic representation. The assignment table has $|GF(2^m)|^n$ entries, each giving a value of $A[f]$ for a different possible input vector. Given two functions f_0 and f_1, a comparison of their function values compares corresponding entries in their truth table. If $f_0 \neq f_1$ we are guaranteed of only one

difference between the truth tables, and only a rather weak guarantee can be given on the probability that a random comparison will find it. On the other hand, by theorem 4.12 we know that the assignment tables of two inequivalent functions must differ in at least $(|GF(2^m)| - 1)^n$ entries. Thus, the difference on a single entry in the truth table is greatly magnified to provide assignment tables that differ in nearly all entries. It is this magnification effect that makes signature based verification so attractive compared to Boolean simulation.
Finally, as an additional extension of the $\oplus$-OBDD data structure let us remark that the probabilistic equivalence test can also be applied to $\oplus$-FBDDs that generalize FBDDs in a similar way as $\oplus$-OBDDs generalize OBDDs.

Definition 4.8 *A $\oplus$-FBDD is defined in the same way as a FBDD with additional $\oplus$-nodes that can be part of the data structure as in the case of $\oplus$-OBDDs. This means that a $\oplus$-FBDD is a directed acyclic graph that consists out of branching nodes, labeled with a Boolean variable $x_i \in \{x_1, \ldots, x_n\}$, $\oplus$-nodes, and the sink nodes. On every path from a root node to the sinks, each variable must occur at most once.*

As in the case of FBDDs, we can extend the definition of $\oplus$-FBDDs also to type consistent $\oplus$-FBDDs (τ-$\oplus$-FBDDs). As $\oplus$-OBDDs, $\oplus$-FBDDs are not a canonical data structure. But, in order to make binary synthesis feasible for $\oplus$-FBDDs, as in the case of FBDDs, we have to consider type restricted $\oplus$-FBDDs.

Definition 4.9 *A $\oplus$-FBDD P defines a $\oplus$-FBDD type T_P in a similar way as FBDDs do. P is called type restricted, if it is possible to transform T_P into an ordinary FBDD type τ_P by means of a sequence of reductions.*

In order to transform an $\oplus$-FBDD type T into a FBDD type τ, we simply eliminate all $\oplus$-nodes starting from the sink and by applying *deletion* and *merging* rule to the remaining type until no further reduction can be performed anymore. Of course, the possibility of such an elimination of an $\oplus$-node does not say anything about the redundancy of these nodes in the original $\oplus$-FBDD.

Definition 4.10 *Two $\oplus$-FBDDs P, Q are called* **consistent**, *if they are type restricted and if the corresponding FBDD types τ_P and τ_Q are consistent.*

But, $\oplus$-OBDDs are not the main subject of this thesis and will not be considered any further. In a next step to support efficient working with $\oplus$-OBDDs, we focus on $\oplus$-OBDD synthesis.

4.5 Synthesis of $\oplus$-OBDDs

For Boolean function manipulation by means of a representation like $\oplus$-OBDDs we are in need of an efficient algorithm that is able to perform the application of an arbitrary binary Boolean operator to two $\oplus$-OBDDs. For the creation of a $\oplus$-OBDD based on a given circuit description, we are constructing the

$\oplus$-OBDD gate by gate starting with the circuit's primary inputs in direction to its outputs. For every single gate G of the circuit description and its inputs $G_{IN1}, \ldots, G_{INm}$ representing a Boolean operation $G = G_{IN1} \otimes G_{IN2} \otimes \ldots \otimes G_{INm}$, a $\oplus$-OBDD P_G is constructed by applying the Boolean operator $\otimes$ to the $\oplus$-OBDDs $P_{G_{IN1}}, \ldots, P_{G_{INm}}$ representing the inputs of G. Thus, P_G is constructed by applying a synthesis procedure $P_G = \bigotimes_{i=1}^{m} P_{G_{INi}}$.
As shown in [GM93] the result of the application of an arbitrary Boolean operator $\otimes$ to two $\oplus$-OBDDs R and Q of the same variable ordering π can be constructed in time $O(|R| \cdot |Q|)$. Even better, if $\otimes \in \{\oplus, \equiv\}$, then the resulting $\oplus$-OBDD can be constructed in constant time.
But, before we describe the different synthesis procedures in detail, we have to take care about necessary prerequisites that have also to be adapted for $\oplus$-OBDDs synthesis.

4.5.1 Cofactor Creation

The already known synthesis procedure for OBDDs (ITE-algorithm) is working recursively based on the Boole/Shannon-expansion of the function f under consideration w.r.t. its variables $x_i \in \{x_1, \ldots, x_n\}$

$$f = \overline{x_i} \cdot f_{\overline{x_i}} + x_i \cdot f_{x_i}.$$

For OBDDs, computing the cofactor w.r.t. the top variable of the OBDD is quite simple. To compute f_{x_i}, with x_i being the top variable of the OBDD P_f, we simply return the successor of the node representing f, where the 1-edge is pointing to. To compute $f_{\overline{x_i}}$ we follow the other edge, respectively. This operation can be performed in constant time $O(1)$. Note that we only consider the case of creating a cofactor f_{x_i} for the top variable x_i. If the top variable of f is another variable x_j, $j \neq i$, then we assume that $j > i$ according to the given variable order π such that x_i does not occur in the $\oplus$-OBDD of f.
For the computation of a cofactor $f_{x_i}(f_{\overline{x_i}})$ w.r.t. the top variable x_i, for $\oplus$-OBDDs we have to distinguish two different cases:

(1) The top node of P_f is a branching node v_B labeled with the variable x_i, or

(2) the top node of P_f is a $\oplus$-node $v_\oplus$.

For case (1), we create the cofactor $f_{x_i}(f_{\overline{x_i}})$ in the same way as for regular OBDDs, i.e. we return its 1-successor for f_{x_i} and its 0-successor for $f_{\overline{x_i}}$, respectively.
For case (2), the top node of f is labeled with a $\oplus$-node. Its two successors $v_{\oplus_r}$ and $v_{\oplus_l}$ are denoting the functions f_l and f_r. If we want to compute $f_{x_i}(f_{\overline{x_1}})$, we have to create a new $\oplus$-node $v'_\oplus$ that has to be connected with the cofactors $f_{l_{x_i}}(f_{l_{\overline{x_i}}})$ of f_l and $f_{r_{x_i}}(f_{r_{\overline{x_i}}})$ of f_r. But, $v'_\oplus$ only has to be created, if the node that is representing f_{x_i} does not already exist in the $\oplus$-OBDD. If a successor $v_{\oplus_l}(v_{\oplus_r})$ of $v_\oplus$ is a branching node, then we proceed as for cofactor creation of regular OBDDs (1). Otherwise, if a successor $v_{\oplus_l}(v_{\oplus_r})$ of $v_\oplus$ is a $\oplus$-node, we

```
Input: ⊕-OBDD P_f, a variable x_i, and the assignment a
Output:⊕-OBDD P_{f_{x_i}} or P_{f_{\overline{x_i}}}.

cofactor(P_f, x_i, pol) {
   if (top node of P_f is branching node) {
     if (top variable of P_f is x_i) {
       if (a == 1) {
         return(1-successor of P_f);
       } else {
         return(0-successor of P_f);
       }
     } else {
       return(P_f);
     }
   } else { // top node of P_f is a ⊕-node
     new_left = cofactor(1-successor of P_f, a);
     new_right = cofactor(0-successor of P_f, a);
     if (node(XOR, new_left,new_right) exists) {
       return(node);
     } else {
       node=create(XOR, new_left, new_right);
       return(node);
     }
   }
}
```

Figure 4.13: Cofactor Creation Algorithm for ⊕-OBDDs in Pseudo Code

continue as for step (2) (See Fig. 4.13 for the cofactor creation algorithm given in pseudo code).
Given a ⊕-OBDD P_f that is representing f, w.r.t. a fixed variable $x_i \in \{x_1, \ldots, x_n\}$ and an assignment $a \in \{0, 1\}$ the algorithm computes the ⊕-OBDD $P_{f_{x_i}}(P_{f_{\overline{x_i}}})$ representing the cofactor $f_{x_i}(f_{\overline{x_i}})$.
For the case that the ⊕-OBDD P_f does not depend on the variable x_i and its top node happens to be a ⊕-node $v_\oplus$, no new ⊕-node has to be created, and P_f is returned. There are two possible ways to detect this particular situation, dependent on the implementation of the ⊕-OBDD data structure:
On one hand we might look for the first branching node among the successors of $v_\oplus$, which is first w.r.t the given variable order π. But, for this solution, if some of the successor nodes of $v_\oplus$ also happen to be ⊕-nodes, again all of their successors have now to be tested, too. On the other hand, we also have the possibility to label each ⊕-node $v_\oplus$ with the label of the first successing branching node w.r.t. π. If we choose the second approach, we can immediately decide, whether P_f is dependent of x_i, also in the case, when the top node of P_f is a ⊕-node. For the implementation of our ⊕-OBDD package we have

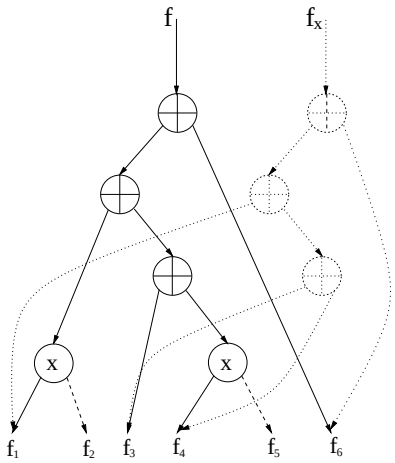


Figure 4.14: Cofactor Creation for $\oplus$-OBDDs.

chosen this solution simply for reasons of efficiency, because cofactor creation is one of the operations that have to be performed most frequently in symbolic simulation.
Thus, the worst case time complexity of the cofactor creation algorithm for $\oplus$-OBDDs is $O(|P|)$, no matter for which variable $x_i \in \{x_1, \ldots, x_n\}$ the restriction is computed. If we consider a worst case scenario and none of the cofactor $\oplus$-nodes to be created do already exist in the $\oplus$-OBDD under consideration, they all have to be newly created. See Fig. 4.14 for an illustrating example of this situation. There, the $\oplus$-OBDD of the cofactor f_x and the $\oplus$-nodes that have to be newly created are denoted by dotted lines.

4.5.2 The Standard Apply Algorithm

The standard manipulation algorithm for $\oplus$-OBDDs is based on the Boole-/Shannon-expansion of the Boolean functions represented by the $\oplus$-OBDDs to be combined. This algorithm, also known as *standard apply algorithm* can be implemented recursively. But, for its application it is important that the Boolean operator to be applied is distributive over the operations represented by the $\oplus$-OBDD nodes, esp. the distributivity over the XOR operator.

Theorem 4.18 *Let P_f, P_g be two $\oplus$-OBDDs representing the Boolean functions $f, g : \{0,1\}^n \rightarrow \{0,1\}$ and let $\otimes$ be an arbitrary Boolean operator. $P_f \otimes P_g$ can be computed by recursive application of the operator $\otimes$ on the successors of P_f and P_g.*

Proof: To show the required property for an arbitrary Boolean operation $\otimes$, it is sufficient to show that the property holds for a complete Boolean operator basis, e.g. $\{AND, XOR\} \equiv \{\cdot, \oplus\}$. We have to distinguish two different cases: the top node of P_f and P_g can either be a branching node or a $\oplus$-node. First, let the top node of P_f and P_g be a branching node labeled with $x_i \in \{x_1, \ldots, x_n\}$.

For $\otimes = \{AND\}$ it holds that

$$\begin{aligned} f \cdot g &= (x_i \cdot f_{x_i} + \overline{x_i} \cdot f_{\overline{x_i}}) \cdot (x_i \cdot g_{x_i} + \overline{x_i} \cdot g_{\overline{x_i}}) \\ &= (x_i \cdot f_{x_i} \cdot x_i \cdot g_{x_i}) + (x_i \cdot f_{x_i} \cdot \overline{x_i} \cdot g_{\overline{x_i}}) + \\ &\quad (\overline{x_i} \cdot f_{\overline{x_i}} \cdot x_i \cdot g_{x_i}) + (\overline{x_i} \cdot f_{\overline{x_i}} \cdot \overline{x_i} \cdot g_{\overline{x_i}}) \\ &= (x_i \cdot f_{x_i} \cdot x_i \cdot g_{x_i}) + 0 + 0 + (\overline{x_i} \cdot f_{\overline{x_i}} \cdot \overline{x_i} \cdot g_{\overline{x_i}}) \\ &= x_i \cdot (f_{x_i} \cdot g_{x_i}) + \overline{x_i} \cdot (f_{\overline{x_i}} \cdot g_{\overline{x_i}}). \end{aligned}$$

and for $\otimes = \{XOR\}$ it holds that

$$\begin{aligned} f \oplus g &= (x_i \cdot f_{x_i} + \overline{x_i} \cdot f_{\overline{x_i}}) \oplus (x_i \cdot g_{x_i} + \overline{x_i} \cdot g_{\overline{x_i}}) \\ &= (x_i \cdot f_{x_i} \oplus \overline{x_i} \cdot f_{\overline{x_i}}) \oplus (x_i \cdot g_{x_i} \oplus \overline{x_i} \cdot g_{\overline{x_i}}) \\ &= (x_i \cdot f_{x_i} \oplus x_i \cdot g_{x_i}) \oplus (\overline{x_i} \cdot f_{\overline{x_i}} \oplus \overline{x_i} \cdot g_{\overline{x_i}}) \\ &= x_i \cdot (f_{x_i} \oplus g_{x_i}) + \overline{x_i} \cdot (f_{\overline{x_i}} \oplus g_{\overline{x_i}}). \end{aligned}$$

Note that for the Boole/Shannon expansion $f = x_i \cdot f_{x_i} + \overline{x_i} \cdot f_{\overline{x_i}}$ the two parts of the sum are disjunctive. Therefore, it is possible to substitute $'+'$ by $'\oplus'$ without changing the function's result.
Now, let the top node of one of the two $\oplus$-OBDDs be a $\oplus$-node, w.l.o.g. let $f = f_l \oplus f_r$ (the other case is symmetric).
For $\otimes = \{AND\}$ it holds that

$$\begin{aligned} f \cdot g &= (f_l \oplus f_r) \cdot (x_i \cdot g_{x_i} + \overline{x_i} \cdot g_{\overline{x_i}}) \\ &= f_l \cdot (x_i \cdot g_{x_i} + \overline{x_i} \cdot g_{\overline{x_i}}) \oplus f_r \cdot (x_i \cdot g_{x_i} + \overline{x_i} \cdot g_{\overline{x_i}}) \end{aligned}$$

and

$$\begin{aligned} f \oplus g &= (f_l \oplus f_r) \oplus (x_i \cdot g_{x_i} + \overline{x_i} \cdot g_{\overline{x_i}}) \\ &= f_l \oplus (x_i \cdot g_{x_i} + \overline{x_i} \cdot g_{\overline{x_i}}) \oplus f_r \oplus (x_i \cdot g_{x_i} + \overline{x_i} \cdot g_{\overline{x_i}}). \end{aligned}$$

Thus, the computation of an arbitrary binary Boolean operator applied to two $\oplus$-OBDDs can be performed by its recursive application to the cofactors of the $\oplus$-OBDDs. □

Based on this theorem we can define a simple algorithm for applying an arbitrary Boolean operator $\otimes$ to two $\oplus$-OBDDs, further denoted as `standard-apply`. First, $\otimes$ has to be expressed in terms of the complete Boolean operator basis $\{AND, XOR\}$. Then, for the AND-operator the AND-synthesis procedure is called, while for the XOR-operator simply a new $\oplus$-node is created and connected to the two operands.
The AND-synthesis procedure is working in the following way:
If one of the $\oplus$-OBDDs P_f, P_g is a 1(0)-sink, the computed result of $f \cdot 1 = f(f \cdot 0 = 0)$ can be directly returned. If the top node of P_f or of P_g is a $\oplus$-node, w.l.o.g. let v be the top node of P_g labeled with $\oplus$, then, we create a new $\oplus$-node and connect it with the results of $(f \cdot v_l)$ and $(f \cdot v_r)$ (see Fig. 4.15 for an illustration). If the top nodes of P_f and P_g are branching nodes that are labeled with the variable x, then we create a new branching node that is labeled with x and connect it with the result of $P_{f_x} \cdot P_{g_x}$ and $P_{f_{\overline{x}}} \cdot P_{g_{\overline{x}}}$.
To avoid *redundant* recursive calls we can make use of cache table $T[x, y]$ of the size $|P_f| \times |P_g|$, which contains possible partial results of $P_f \cdot P_g$. All entries of

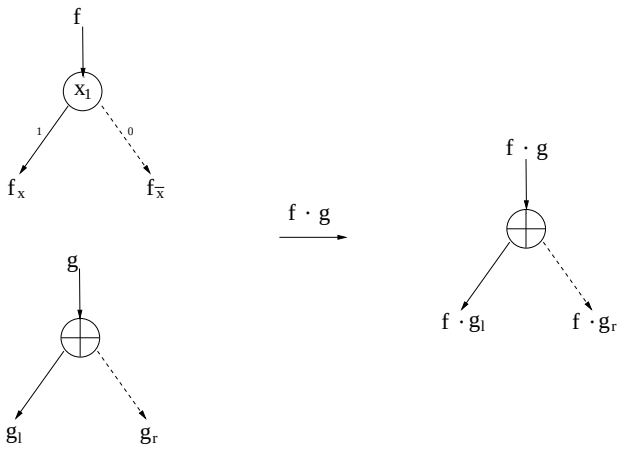


Figure 4.15: AND-Synthesis with the `standard-apply` Algorithm for $\oplus$-OBDDs.

T are initialized with zero. Before a new node is ceated, the table T is tested, if this node has been already computed. If the node is found in T, then we can directly return the node. Otherwise, the node is created according to the given procedure and afterwards, it is stored within T. See Fig 4.16 for an outline of the `standard-apply` algorithm for $\oplus$-OBDDs in pseudo code.
For the implementation of the complementation $\overline{f}$ of the Boolean function f we can take advantage of the rule $\overline{f} = f \oplus 1$, i.e. we create a new $\oplus$-node and connect it with the top node of the $\oplus$-OBDD representing f and with the 1-sink.
The correctness of the algorithm follows from the easily provable correctness of each of its recursive steps. Since the number of recursive calls is bounded by the size of T, the time complexity of the `standard` $-$ `apply` algorithm is bounded by $O(|P_f| \cdot |P_g|)$.
Instead of using $\{AND, XOR\}$ as a basis for the $\oplus$-synthesis-procedure we might also adapt the well known ITE-algorithm that is applied in most OBDD packages of commercial relevance, because of its efficiency.

4.5.3 The ITE-$\oplus$ Algorithm

When adapting the ITE-algorithm for OBDDs to be used for $\oplus$-OBDDs, we have to take care about the following major differences:

(1) The $\oplus$-operation can be directly mapped to the creation of a new $\oplus$-node, and

(2) for recursively calling the ITE-procedure the cofactor creation has to be adapted for $\oplus$-OBDDs.

The standard ITE-algorithm for OBDDs is based on the ternary operator ITE:

$$ite(f, g, h) = f \cdot g + \overline{f} \cdot h$$

Input: $\oplus$-OBDDs P_f, P_g, and an arbitrary Boolean operator op.
In a preprocessing step op *is mapped to the complete basis* $\{AND, XOR\}$ *and the operands* P_f *and* P_g *are adapted, respectively.*

Output:$\oplus$-OBDD P_h, representing the Boolean function $h = f\ op\ g$.

```
standard-apply(P_f, P_g, op) {
  if(T[P_f,P_g] ≠ 0) {
    return (T[P_f,P_g]);
  } else {
    if (op == ⊕) {
      if (node(⊕, P_f, P_g) does not already exist) {
        create node(⊕, P_f, P_g);
      }
      T[P_f,P_g] = a pointer to node(⊕, P_f, P_g);
      return(node(⊕, P_f, P_g));
    } else {
      if (P_f or P_g is a 1-sink ) {
        T[P_f,P_g] = a pointer to the other ⊕-OBDD;
        return the other ⊕-OBDD
      } else if (P_f or P_g is a 0-sink ) {
        T[P_f,P_g] = a pointer to the 0-sink;
        return the 0-sink
      }
      if (root of P_f or P_g is a ⊕-node) {\\here w.l.o.g. source of P_g
        new_left = standard-apply(P_f, P_{g_l}, op);
        new_right = standard-apply(P_f, P_{g_r}, op);
        if (node(⊕,new_left,new_right) does not already exist) {
          new = create node(⊕, new_left, new_right);
        }
        T[P_f,P_g] = new;
        return(new);
      } else {\\roots of P_f, P_g are branching nodes
        label = min(l(root node(P_f)),l(root node(P_g)));
        new_left = standard-apply(P_{f_label}, P_{g_label}, op);
        new_right = standard-apply(P_{f_label}, P_{g_label}, op);
        if (node(label,new_left,new_right) does not already exist) {
          new = create node(label, new_left, new_right);
        }
        T[P_f,P_g] = new;
        return(new);
      }
    }
  }
}
```

Figure 4.16: `standard-apply` Algorithm for $\oplus$-OBDDs.

and the algorithm can be recursively computed based on the Boole/Shannon expansion:

$$ite(f, g, h) = (x, ite(f_x, g_x, h_x), ite(f_{\overline{x}}, g_{\overline{x}}, h_{\overline{x}})).$$

The adapted ITE-algorithm - from now on denoted as *ITE-⊕-algorithm* - is working in the following way: Let P_a, P_b be two ⊕-OBDDs representing the Boolean functions $a, b : \{0,1\}^n \rightarrow \{0,1\}$, and let $res = a \otimes b$ be the Boolean function to be computed, which in a preprocessing step is transformed into a call to $ite\text{-}\oplus(P_f, P_g, P_h)$. Let π be the variable order of P_a, P_b, P_f, P_g, and P_h. If $\otimes = XOR$, we simply create a new ⊕-node, connect it with the two operands $P_a \oplus P_b$, and return the new ⊕-node as the result. Note that this step is identical to the `standard-apply` algorithm. For reasons of efficiency we also maintain a *unique table* that is implemented as a hash table with collision lists for preventing the creation of nodes that do already exist. In the same way as the regular ITE-algorithm, ITE-⊕ requires three ⊕-OBDDs P_f, P_g, and P_h as input parameters. In addition to the unique table we are maintaining a computed table that is structured like a hash based cache for storing already computed results of the *ITE-⊕* operation. If the result of ITE-$\oplus(P_f, P_g, P_h)$ is already present in the computed table, we simply return the stored result. Otherwise, we are recursively calling ITE-$\oplus(P_{f_x}, P_{g_x}, P_{h_x})$ and ITE-$\oplus(P_{f_{\overline{x}}}, P_{g_{\overline{x}}}, P_{h_{\overline{x}}})$ with the cofactors of P_f, P_g, and P_h for computing the two new successors of the resulting node *res*. The node *res* is labeled with x, which is the top most variable of P_f, P_g, and P_h w.r.t. π. Then, if all successors of *res* have been recursively determined and if *res* does not already exist, it has to be newly created. At last, we store the new node *res* and the operands P_f, P_g, and P_h in the computed table.

The algorithm terminates, if the first of its parameters is a terminal node or another terminal case of the standard ITE-algorithm is accomplished. See Fig. 4.17 for an outline of the *ITE-⊕*-algorithm in pseudo code.

For a more efficient implementation and for increasing the hit rate of the maintained cache, the triple (P_f, P_g, P_h) is transformed into a standard form and reordered in the same way as in the original ITE-algorithm, so that the ⊕-OBDD with the top most root variable according to π is first in the triple. If the implementation supports complemented edges, the algorithm also takes care that no 1-edge of a branching node will be complemented.

The implementation of the computed table as a hash based cache requires entries of the form $((P_f, P_g, P_h)|P_{res})$. Note that the identification of the ⊕-OBDDs (P_f, P_g, P_h) is achieved via Boolean signatures, while P_{res} is the memory address of the resulting ⊕-OBDD.

Due to the use of the computed table the time complexity of the ITE-⊕-algorithm is bounded by the size of the participating ⊕-OBDDs $O(|P_f| \cdot |P_g| \cdot |P_h|)$. But, if we translate an arbitrary binary Boolean operator $P_a \otimes P_b$ into its ITE-equivalent, one of the three parameters of ITE-⊕ will be a Boolean constant. Thus, the time complexity reduces to $O(|P_a| \cdot |P_b|)$. Even better, if $\otimes \in \{\oplus, \equiv\}$, then the synthesis can be computed in constant time, because $(f \equiv g) = (f \oplus g \oplus 1)$.

Input: $\oplus$-OBDDs P_a, P_b, and an operator op.
(P_a op P_b) will be preprocessed in PRE-ITE-$\oplus$
Output:$\oplus$-OBDD P_{res} representing the Boolean function $h = f\ op\ g$.

```
ITE-⊕(P_f, P_g, P_h) {
  transform_to_standard_triple(P_f, P_g, P_h);
  if (terminal_case(P_f, P_g, P_h)) {
    return(P_res);
  }
  reorder triple acc to variable order(P_f, P_g, P_h);
  check rules for complemented edges(P_f, P_g, P_h);
  if (in computed table(P_f, P_g, P_h)) {
    return(P_res);
  } else {
    x = top-variable(P_f, P_g, P_h);
    new_left=ITE-⊕(P_fx, P_gx, P_hx);
    new_right=ITE-⊕(P_fx̄, P_gx̄, P_hx̄);
    if (signature(new_left)==signature(new_right)) {
      P_res=new_left;
    } else {
      P_res=create_node(x,new_left,new_right);
    }
    insert_in_computed_table(P_f, P_g, P_h, P_res);
  }
  find or add in unique table(P_res);
  return(P_res);
}

PRE-ITE-⊕(P_f, P_g, op) {
  if (op == XOR) {
    P_res=create_node(⊕, P_f, P_g);
    find or add in unique table(P_res);
  } else {
    transform (P_f, P_g) into ITE triple (P_f, P_g, P_h);
    P_res=ITE-⊕(P_f, P_g, P_h);
  }
  return(P_res);
}
```

Figure 4.17: The ITE-$\oplus$ Algorithm for $\oplus$-OBDD Synthesis.

4.5.4 Extending the Synthesis Algorithm

In symbolic synthesis by applying the ITE-$\oplus$-algorithm we are translating all gates of a given circuit description via its ITE-$\oplus$-equivalent into a $\oplus$-OBDD starting from the primary inputs to the circuit outputs. Each gate that is representing the Boolean function $f \in \{\equiv, \oplus\}$ in the circuit description will be directly mapped into a $\oplus$-node or into its complement. But, for every other gate that is representing a Boolean function $f \notin \{\equiv, \oplus\}$ the ITE-$\oplus$-algorithm is working exactly in the same way as the regular ITE-algorithm for OBDDs and thus, no new additional $\oplus$-nodes will be introduced into the $\oplus$-OBDD.
So, what if a given circuit description does not contain any gate that is representing $f \in \{\equiv, \oplus\}$? The ITE-$\oplus$-algorithm will work like the regular ITE-algorithm and an OBDD will be created instead of a $\oplus$-OBDD.
Thus, we have to find new ways of introducing additional $\oplus$-nodes into the $\oplus$-OBDD that is to be synthesized. One way of doing this independently of the function being represented as an $\oplus$-OBDD is deploying decompositions for Boolean functions, which, in difference to the used Boole/Shannon decomposition are explicitely using the $\oplus$-operator. These $\oplus$-operators can be directly mapped into new $\oplus$-nodes in the $\oplus$-OBDD. The two alternative function decompositions under further consideration are the positive and negative Davio decomposition (expansion) (pDE/nDE):

$$\begin{aligned} \text{pDE:} \quad & f = f_{\overline{x}} \oplus x \cdot (f_{\overline{x}} \oplus f_x) \\ \text{nDE:} \quad & f = f_x \oplus \overline{x} \cdot (f_{\overline{x}} \oplus f_x) \end{aligned}$$

Other decomposition types can be neglected, as it is shown in [BD95]. If we consider the application of an arbitrary binary Boolean operator $\otimes$ to two Boolean functions $f, g : \{0,1\}^n \rightarrow \{0,1\}$, $h = f \otimes g$, we can expand h with one of the above given function decompositions by the variable $x \in \{x_1, \ldots, x_n\}$. W.l.o.g. we will use pDE:

$$h = f \otimes g = (f_{\overline{x_i}} \otimes g_{\overline{x_i}}) \oplus x_i \cdot ((f_{\overline{x_i}} \otimes g_{\overline{x_i}}) \oplus (f_{x_i} \otimes g_{x_i}))$$

Now, we are able to extend the `standard-apply` algorithm for $\oplus$-OBDDs that is based on Boole/Shannon decomposition to Davio decomposition, where we are able to introduce two new $\oplus$-nodes with each decomposition step (see Fig. 4.18). Note that it is also possible to use the Boole/Shannon-decomposition based on the XOR-operator

$$f = x_i \cdot f_{x_i} \oplus \overline{x_i} \cdot f_{\overline{x_i}}$$

or to map every possible Boolean operator of the circuit description to the complete bases (AND, XOR). But, the usage of the Davio decompositions has been proven to be more efficient.

The extended `apply`-algorithm for $\oplus$-OBDDs - from now on denoted as `apply`-$\oplus$ - differs only the synthesis part. Further, w.l.o.g. we will consider pDE. First, we call the algorithm recursively for the positive and negative cofactors of the operands $P_{f_1} =$ `apply`-$\oplus(P_{f_x}, P_{g_x}, op)$ and $P_{f_0} =$ `apply`-$\oplus(P_{f_{\overline{x}}}, P_{g_{\overline{x}}}, op)$. Then, a new $\oplus$-node is created and connected to P_{f_1} and P_{f_0}.

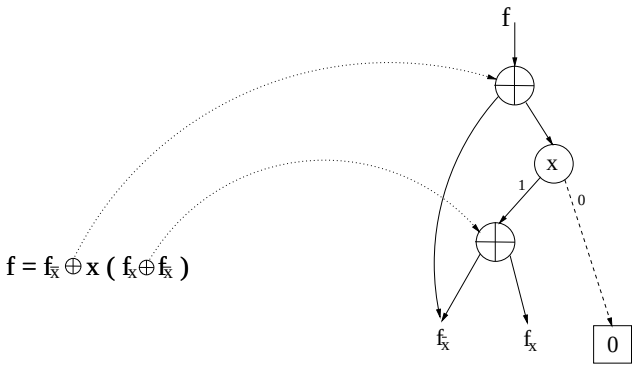


Figure 4.18: Transformation from pDE-Formula to ⊕-OBDD.

Next, the `apply`-⊕ algorithm is called for computing the product $x \cdot (f_0 \oplus f_1)$, which forms the ⊕-OBDD P_x. The last step is the creation of a second ⊕-node that is connected with P_{f_0} and P_x. Note that new nodes are only created, if they do not already exist and that also a computed table is maintained for avoiding redundant computations. See Fig 4.19 for an outline of the `apply`-⊕ algorithm in pseudo code.
For each recursion step we introduce at most two new ⊕-nodes. Obviously many of the ⊕-nodes that are created in this way are redundant and reduction rules can be applied.
Note that in Fig. 4.19 we have mapped the Boolean operator to the complete basis (AND, XOR). Otherwise, we have to support caches for each single operator, or we have to use an extended cache table $T[P_f, P_g, op]$ for storing already computed results.
As we will show in the upcoming chapter that is dedicated to the minimization of ⊕-OBDDs, number and placement of ⊕-nodes within the ⊕-OBDD are important factors that determines size of the resulting ⊕-OBDD. We have the possibility to combine all different synthesis algorithms given so far and thus, we are in control of how often pDE or nDE will be applied for ⊕-node creation.

4.6 Basic Manipulation Tasks for ⊕-OBDD

We have shown the advantages of the ⊕-OBDD data structure in comparison to other decision diagram types and have pointed out the existence of exponential gaps in their representational power. But, for working efficiently with this new data structure the algorithms that are performing the basic manipulation tasks have to be of polynomial complexity. As mentioned before, by being a non canonical data structure some of the manipulation tasks for ⊕-OBDDs, in particular those that are related to testing the equivalence of two ⊕-OBDDs, are not simple to perform. In the following theorem we have summarized the already mentioned results of the complexity evaluation of basic manipulation tasks for ⊕-OBDDs:

Theorem 4.19 *Let G, G_1, and G_2 be ⊕-OBDDs ordered w.r.t. a variable*

```
Input: ⊕-OBDDs P_f, P_g, and an operator op ∈ {AND, XOR}
Output:⊕-OBDD P_h representing the Boolean function h = f op g.

apply-⊕(P_f, P_g, op) {
  if (P_f or P_g is a sink ) {
    res = directly computed result of op;
    T[P_f, P_g] =res;
    return(res);
  }
  if(T[P_f, P_g] ≠ 0) {
    return (T[P_f, P_g]);
  } else {
    if (op == ⊕) {
      if (node(⊕, P_f, P_g) does not already exist) {
        create node(⊕, P_f, P_g);
      }
      res = a pointer to node(⊕, P_f, P_g);
      T[P_f, P_g] = res;
      return(res);
    } else {
      x = top variable of (P_f, P_g);
      new_x= apply-⊕(P_{f_x}, P_{g_x}, op);
      new_x̄= apply-⊕(P_{f_x̄}, P_{g_x̄}, op);
      if (node(⊕, new_x, new_x̄) does not already exist) {
        create node(⊕, new_x, new_x̄);
      }
      cof = node(⊕, new_x, new_x̄);
      P_x = ⊕-OBDD representing the variable x;
      left = apply-⊕(P_x, cof, AND);
      if (node(⊕, new_x̄, left) does not already exist) {
        create node(⊕, new_x̄, left);
      }
      res = node(⊕, new_x̄, left);
      T[P_f, P_g] = res;
      return (res);
    }
  }
}
```

Figure 4.19: apply-⊕ Synthesis Algorithm for ⊕-OBDDs Based on pDE.

ordering π representing the Boolean functions $g, g_1, g_2 : \{0,1\}^n \to \{0,1\}$, let $x_i \in \{x_1, \ldots, x_n\}$ be an arbitrary variable, and let $c \in \{0,1\}$ be a binary Boolean constant.

(1) The evaluation of G can be performed in time $O(|G|)$.

(2) Replacement of an arbitrary variable x_i in G by a constant c, $G_{x_i=c}$ can be computed in time $O(|G|)$. The resulting graph G' is again ordered by π and it holds that $|G'| \leq |G|$.

(3) Equivalence of G_1 and G_2 can be decided probabilistically in time $O(|G_1| + |G_2|)$. Note that testing the equivalence of two Boolean functions g_1 and g_2 is equivalent to the satisfiability problem for $g_1 \oplus g_2$.

(4) Satisfiability of G can be tested probabilistically in time $O(|G|)$, simply by testing the equivalence of G and the constant function 0.

(5) Boolean synthesis $G_1 \otimes G_2$, with $\otimes$ being an arbitrary binary Boolean operator can be computed in time $O(|G_1| \cdot |G_2|)$.

(6) Complementation of the Boolean function $\overline{g}$ can be computed in constant time $O(1)$.

(7) Universal quantification $\forall x_i : g$ and existential quantification $\exists x_i : g$ can be computed in time $O(|G|^2)$.

(8) Composition of G_1 by replacement of an arbitrary variable x_i with G_2 $(g_1|_{x_i=g_2})$ can be computed in time $O(|G_1|^2 \cdot |G_2|)$.

Proof: (1)-(6) have been shown in the previous chapters.
(7) By computing $\forall x_i : g = g_{x_i} \cdot g_{\overline{x_i}}$ and $\exists x_i : g = g_{x_i} + g_{\overline{x_i}}$ with (5) we are achieving the assumed result.
(8) Since restriction of a $\oplus$-OBDD by a partial variable assignment can be computed in linear time (2) we can compute the $\oplus$-OBDD for $g_1|_{x_i=g_2}$ with the Boole/Shannon decomposition as

$$g_1|_{x_i=g_2} = \overline{g_2} \cdot g_{1_{\overline{x_i}}} + g_2 \cdot g_{1_{x_i}} = \text{ite-}\oplus(g_2, g_{1_{x_i}}, g_{1_{\overline{x_i}}}).$$

and yield a time complexity of $O(|G_1|^2 \cdot |G_2|)$. □

4.7 Applying $\oplus$-OBDDs in Symbolic Simulation

For demonstrating the efficiency of $\oplus$-OBDDs a programming package that realizes the basic tasks of symbolic simulation based on $\oplus$-OBDDs has been developed. Before the package could be deployed for experiments in symbolic simulation, the reliability of the used data structure had to be proven, because $\oplus$-OBDD based symbolic simulation is relying on the application of a probabilistic equivalence test. Thus, we had to investigate, how many signatures of which length were sufficient for avoiding the possibility of errors.

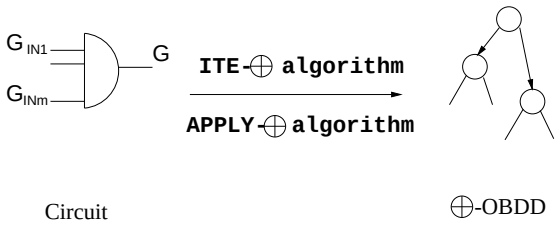


Figure 4.20: Symbolic Simulation with ⊕-OBDDs.

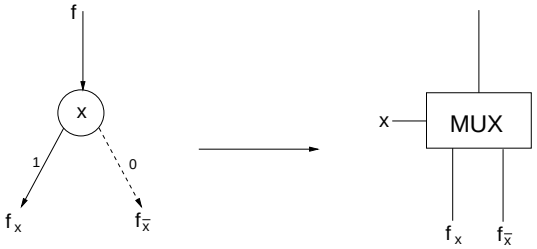


Figure 4.21: Equivalence of Branching Node and Multiplexer Gate.

4.7.1 Experimental Setup

The basic task of symbolic simulation is the translation of a given circuit description into a specific data structure, which can be utilized for further manipulation tasks to be computed on the circuit description. The circuit description usually is specified in an appropriate design language in form of a text file. It contains identifiers for the circuit's primary inputs (PI), the gates, all the interconnections, and the primary outputs (PO). For sequential circuits, loops, feedback lines, and latches are broken into separate inputs and outputs. Then, starting with the PIs, each successing gate is translated into the target data structure - here, a ⊕-OBDD. To translate the gate G, depending on the $m \in \mathbb{N}$ inputs $G_{IN1}, \ldots, G_{INm}$ and computing the Boolean function $G = \bigotimes_{i=1}^{m} G_{INi}$, successive calls to one of the introduced synthesis operations (ITE-⊕, apply-⊕) have to be performed (see Fig. 4.20).

But, working with ⊕-OBDDs and thus, depending on a probabilistic equivalence test necessitates the possibility to proof the reliability of the approach. For that reason, the constructed ⊕-OBDD again was retranslated into a textual circuit description. This retranslation can be performed by regarding every node of the ⊕-OBDD as either a multiplexer element (see Fig. 4.21), if the node is a branching node, or as a ⊕-gate, if the node is a ⊕-node, respectively. Then, this new circuit description $C_{\oplus-OBDD}$ can be verified against the original given circuit description $C_{Circuit}$ with a standard verification tool, as e.g. VIS [VIS96]. The verification tool is translating both, $C_{\oplus-OBDD}$ and $C_{Circuit}$ into an internal symbolic representation and the equivalence of both is tested. If, during the construction of the ⊕-OBDD the probabilistic equivalence test is working without any errors, the result of the verification tool must be positive. See Fig. 4.22 for an illustration of this approach.

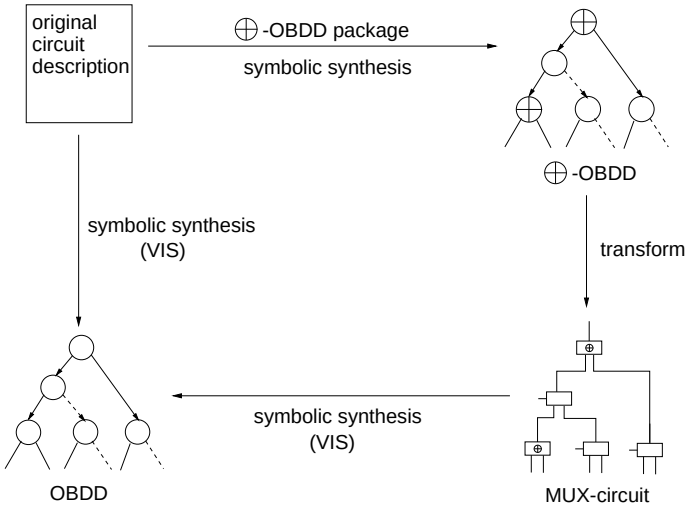


Figure 4.22: Experimental Setup for Testing the Reliability of ⊕-OBDDs.

# of signatures	1	2	3	4
# correct/ # total circuits	32/42	42/42	42/42	42/42

Table 4.4: Testing Signature Reliability.

4.7.2 Experimental Results

For our experimental setup we have chosen a subset of the benchmark sets of LGSynth'93 [LGS]. The benchmark set consists out of combinatorial and sequential examples of wide variety in size and functionality. All experiments were computed on an Intel Pentium III 500 MHz based Linux system. Memory size is limited to 200 MB and computation time to two cpu hours. All circuits of LGSynth'93 that could not be computed within the given resource limits have been excluded. Also excluded are circuits that can be simulated by OBDDs or ⊕-OBDDs of less than 100 nodes. For a comparison, all benchmark circuits are tested with a regular OBDD package - here, the CUDD package- and with our ⊕-OBDD package. In this first round of experiments we are working with fixed variable orders that were computed from the given circuit topology.
The first experiment should give an answer to the question *"How many signatures are necessary"* for achieving reliable results when working with ⊕-OBDDs. We did symbolic simulation of the combinatorial circuits of our complete benchmark set with an increasing number of 32-bit signatures, while verifying the results in the way we have described.
In Table 4.4, for different numbers of signatures the number of circuits that were computed correctly in relation to the total number of benchmark circuits is given.
As a result of this experiment we have fixed a number of $n = 3$ signatures that have been proven to be more than sufficient for all circuits of the benchmark subset. While using only $n = 1$ 32-bit signature, 32 out of 42 circuits, all of

them creating $\oplus$-OBDDs larger than approximately 100000 nodes, contained errors and defects due to signature degeneration. By using $n = 2$ signatures instead, all of our combinatorial benchmark circuits are translated into error-free $\oplus$-OBDDs up to a size of about 10^6 nodes. All further experiments with the $\oplus$-OBDD package have been computed with $n = 3$ signatures of 32-bit length for reasons of security.
Now, in the second set of experiments we want to compare the efficiency of OBDDs and $\oplus$-OBDDs for a given fixed variable order in symbolic simulation. In Table 4.5 significant results for combinatorial and sequential circuits are listed. For the complete results of the benchmark set see Table A.1 and A.2 in the Appendix A.
Many of the circuits in the chosen benchmark set do not contain any XOR-gate. Thus, a plain simulation based on the standard `apply`-$\oplus$-algorithm or ITE-$\oplus$-algorithm results only in an OBDD of size equal to the result of the OBDD package. Therefore, we also conducted all the experiments for the extended `apply`-$\oplus$-algorithm with exclusive application of nDE and pDE, respectively. The results of these experiments denoting the size of the data structures are given in number of nodes. Note that for $\oplus$-OBDDs branching nodes and also $\oplus$-nodes have to be counted. For $\oplus$-OBDDs the first number gives the total size, while the number in braces gives the number of included $\oplus$-nodes, if there are any.
In Table 4.5 the first column denotes the circuit name. The second column gives the size for OBDD simulation, and column 3 to 6 denote the $\oplus$-size for exclusive application of ITE-$\oplus$, and `apply`-$\oplus$ with pDE and nDE, and also for the standard `apply`-$\oplus$ algorithm. The last row contains the overall sum of nodes, computed for the whole benchmark set of 39 combinatorial and 28 sequential circuits. Note that the sum is only given, when all circuits could be computed within the given resource limitations.
Among all 67 circuits there are only a few that contain explicit XOR gates. But, sometimes these few XOR gates are heavily affecting the size of the resulting $\oplus$-OBDD. E.g., if we consider the combinatorial circuits *my_adder, adder16, C499*, and *mult16a*. These circuits contain about 30 XOR-gates, but the reduction capability results in $\oplus$-OBDD sizes that are ranging from 15% to 80% of the original OBDD size. The most impressive result here is $C499$, where the OBDD size of 45922 nodes is reduced to a $\oplus$-OBDD size of only 7030 nodes, containing only 32 $\oplus$-nodes. In all other cases for the ITE-$\oplus$-algorithm the $\oplus$-OBDD size is almost equal to the OBDD size, because in most circuits no new $\oplus$-nodes are created.
Similar is the situation for the `standard apply`-$\oplus$-algorithm. There, for the circuits *my_adder* and $mult16a$ the original OBDD size could be reduced to a $\oplus$-OBDD size of 70%-80%. $C499$, $adder16$, and also 10 other circuits of the combinatorial benchmark set could not be completely computed with `standard-apply`-$\oplus$, because of the given resource limitations. The results for these circuits are denoted by a dash in the table. Compared to the ITE-$\oplus$-algorithm, the `standard apply`-$\oplus$-algorithm requires more recursive calls and its cache efficiency is worse. In the case of the circuit *pair* the result of the `standard apply`-$\oplus$-algorithm requires up to three times the size as the original OBDD,

circuit	size								
	OBDD	**⊕-OBDD**							
		ITE	(⊕)	**nDE**	(⊕)	**pDE**	(⊕)	**stan**	(⊕)
vda	4345	4345		1954	(1399)	1895	(1276)	4345	
my_adder	327677	262188	(30)	589831	(327674)	524297	(262139)	262188	(30)
mux	131071	131071		217	(184)	217	(173)	131071	
i9	2278	2278		8754	(5132)	10258	(6509)	2278	
i2	335	335		317	(47)	851	(581)	335	
frg2	6471	6472	(1)	5031	(2655)	7246	(4451)	6461	
apex1	28336	28336		8901	(6002)	10545	(6930)	28336	
adder16	327812	262310	(32)	606303	(376808)	589904	(360409)	-	
C499	45922	7030	(32)	13699	(10074)	13704	(9829)	-	
C432	1733	1733	(1)	4913	(3154)	4589	(2885)	-	
C1355	45922	45922		14162	(10538)	14167	(10293)	45922	
mult16a	360442	262188	(31)	655125	(392968)	655077	(392920)	262188	(31)
s820	2651	2651		552	(328)	718	(418)	786	
s713	1352	1352		3554	(2114)	3122	(1626)	1056	
s510	19076	19076		636	(433)	739	(503)	1038	
s420	262227	262227		732	(431)	471	(201)	250	
s3384	748809	748809		1142383	(781005)	1139458	(762028)	748809	
s208	1033	1033		186	(105)	141	(67)	80	
s1512	18896	18912		10941	(7295)	7148	(3499)	2621	
s1269	48176	48177		39922	(27974)	33418	(23231)	37459	
Σ	**5.688.571**	n.a.		**6.353.061**		**5.002.668**		n.a.	
	100%			111.7%		87.9%			

Table 4.5: Comparing OBDD and ⊕-OBDD Size for Fixed Variable Order - Combinatorial and Sequential Circuits.

circuit	**Overal Runtime**				
	OBDD	**⊕-OBDD**			
		ITE	**nDE**	**pDE**	**stan**
combinatorial	76.16	n.a.	2440.5	1818.47	n.a.
sequential	22.86	n.a.	962.5	1033.0	n.a.
Σ	99.02	n.a.	3403	2851.47	n.a.
normalized	**1**		34.4	28.8	

Table 4.6: Comparing OBDD and ⊕-OBDD Overall Runtime in CPU-Seconds for Fixed Variable Order - Combinatorial and Sequential Circuits.

while the $\oplus$-OBDD created with ITE-$\oplus$ is about the size of the OBDD.
More interesting are the results for the extended `apply`-$\oplus$-algorithm using pDE and nDE. For the simulated circuits the achieved $\oplus$-OBDD sizes are ranging from almost 0% to 450% of the original OBDD size. In detail, there are $cm150a$, mux, and $s420$ with an OBDD size ranging from 100000 to 200000 nodes that can be represented with $\oplus$-OBDDs of only a few hundred nodes. Also for the circuits $s510$, $mm9a$, $mm9b$, $alu32$, and $alu32r$ the achieved $\oplus$-OBDD size is less than 10% of the original OBDD size. Here, the usage of $\oplus$-nodes reduces the size of the representation by several orders of magnitude and thus, shows a dramatic impact.
But, also if we compare the results for the exclusive application of nDE and pDE only, we can spot out significant differences. For $i2$, $i9$, and also $frg2$ the resulting size of the $\oplus$-OBDDs for nDE and pDE differs from 118% up to 268% for each pair. Also in the sum over all circuits nDE and pDE differ from 87.9% to 111.7% of the original OBDD size. In each recursive step of the extended `apply`-$\oplus$ algorithm up to two new $\oplus$-nodes are created. By considering the differences in size for nDE and pDE, we can conclude that not only the number of included $\oplus$-nodes, but also their placement seems to be significant for the $\oplus$-OBDD size.
For $i2$ the extended `apply`-$\oplus$ algorithm with nDE computes a $\oplus$-OBDD of 94% the size of the original OBDD, while with pDE the $\oplus$-size is 254% of the original size. For nDE, 15% of the nodes of $i2$ are $\oplus$-nodes, while for pDE, 68% of all the nodes are $\oplus$-nodes. Here, the difference in size for sure is related to the portion of $\oplus$-nodes in the $\oplus$-OBDD. If in the extended `apply`-$\oplus$ algorithm pDE and nDE are used exclusively, as it is the case, often too many $\oplus$-nodes are created and thus, affecting the $\oplus$-OBDD size sometimes also in a negative way. Therefore, a more sophisticated algorithm should limit the use of $\oplus$-node producing decompositions during synthesis.
The computation time for symbolic simulation with $\oplus$-OBDDs is difficult to compare with the computation time of an optimized professional OBDD package like CUDD. The factors that are responsible for that fact are the following:

- For every node that has to be created in the $\oplus$-OBDD package a fixed number of Boolean signatures has to be computed, because the node has to be identified with these signatures.

- The cofactor creation for $\oplus$-OBDDs is a much more complex task compared to the creation of cofactors for OBDDs, because it requires recursive calls and special treatment for $\oplus$-nodes. Note that the runtime complexity of the cofactor creation for $\oplus$-OBDDs is bounded by $O(|G|)$ as mentioned in the previous chapter, and that for OBDDs the cofactor can be computed in constant time $O(1)$, if we consider cofactor creation w.r.t. the top variable of the OBDD (as it always is the case in OBDD synthesis).

- Optimized OBDD packages use a sophisticated memory management strategy that minimizes the number memory allocation calls and influences the runtime drastically.

As shown in Table 4.6 the overall runtime for $\oplus$-OBDD synthesis ranges from 28.8 to almost 35 times the time required for OBDD synthesis with the state-of-the-art OBDD package. For `ite`-$\oplus$ and `standard-apply`-$\oplus$ the overall sum is not available since not all computations could be finished within the given resource limitations. For the circuits that always could be completely computed, for `ite`-$\oplus$ the runtime is about the same as in the case for pDE and nDE, but for `standard-apply`-$\oplus$ the runtime was even much worse in many cases. Due to the factors mentioned above and due to the fact that up to now no optimization has taken place a runtime comparison between the two packages is not really fair so far. But, as for some circuits we could gain a reduction in size of several orders of magnitude, their runtime also showed an advantage compared to OBDD runtime. Up to now, it is important to state that although even no optimization has taken place, $\oplus$-nodes might lead to a significant reduction in size of the data structure. This gain in size gives $\oplus$-OBDDs the advantage of being applicable to circuits that are not manageable by standard OBDD methods, because they might exceed the given resource limitations.
For a summary of the achieved results, we are able to give the following statements that are motivating further research presented in the next chapters:

- The use of $\oplus$-nodes can have a dramatic effect on $\oplus$-OBDD size and often leads to a reduction in size.
- Sometimes, the exclusive use of the extended `apply`-$\oplus$ algorithm leads to the creation of too many $\oplus$-nodes and thus, can also affect $\oplus$-OBDD size in a negative way.
- Differences in $\oplus$-OBDD size depending on nDE or pDE lead to the assumption that besides $\oplus$-node frequency, also $\oplus$-node placement is important for the $\oplus$-OBDD size.
- Manipulation of $\oplus$-OBDDs is more time expensive than manipulation of OBDDs.

The focus of the next chapter lies on the optimization of the $\oplus$-OBDD data structure.

Chapter 5

Minimization of $\oplus$-OBDDs

In this chapter we show how to improve the efficiency of the $\oplus$-OBDD data structure. Improving the efficiency for $\oplus$-OBDDs means, reducing their overall size. In difference to OBDDs, The size of a $\oplus$-OBDD depends on several factors: First, as for OBDDs, the chosen variable order is an important factor for $\oplus$-OBDD size. Additionally, the number of $\oplus$-nodes in the diagram, and also the placement of the $\oplus$-nodes inside the $\oplus$-OBDD play an important role. Improving the variable order π alone is a **NP**-hard problem. Thus, taking into account that $\oplus$-node position and number has to be optimized separately, the overall minimization of $\oplus$-OBDDs is even more difficult and the only practical way to solve the problem is the development of appropriate heuristics for the single optimization tasks. First, we take a look on the influence of the number of $\oplus$-nodes inside a given $\oplus$-OBDD, before we investigate the importance of the placement of $\oplus$-nodes. Since, our goal is to achieve a small $\oplus$-OBDD P_f for a given Boolean function f, we have to investigate the effects of both, the number of $\oplus$-nodes used inside a $\oplus$-OBDD, and their placement, respectively. While additional $\oplus$-nodes can be introduced by using one of the alternative function decompositions (pDE, nDE), their placement can either be decided by determining a well suited point during $\oplus$-OBDD synthesis for switching from standard decomposition (BS) to a decomposition that is introducing additional $\oplus$-nodes (pDE, nDE), or the chosen positions of $\oplus$-nodes can be adjusted afterwards by dynamic relocation of already created $\oplus$-nodes inside the given $\oplus$-OBDD, which will be the topic of the upcoming sections.

Questions of efficient implementation technique are risen and we present our solution that is based on merging chains and trees of $\oplus$-nodes into so called meta-$\oplus$-nodes. By using these meta-$\oplus$-nodes, the exchange of adjacent variables becomes much easier to implement and constitutes the basis of our version of the variable reordering for $\oplus$-OBDDs. Finally, we develop and evaluate heuristics for $\oplus$-OBDD optimization on the basis of these fundamental techniques.

5.1 $\oplus$-Node Frequency

5.1.1 Prerequisites

By introducing functional operator nodes as in the case of $\oplus$-OBDDs, the efficiency of the OBDD data structure can be enhanced by simultaneously giving up canonicity. If we consider a $\oplus$-OBDD P_f representing the Boolean function $f = f_l \oplus f_r$, this function can be directly translated into a $\oplus$-OBDD by creating a $\oplus$-node v and connecting it with the $\oplus$-OBDDs P_{f_l} and P_{f_r} for f_l and f_r. Thus, the size of P_f is bounded by merely $O(|P_{f_l}| + |P_{f_r}|)$.
On the other hand, instead of introducing a new $\oplus$-node, we can translate $f = f_l \oplus f_r$ to a call of the ITE-algorithm.

$$\begin{aligned} f &= f_l \oplus f_r \\ &= (f_l \cdot \overline{f_r}) + (\overline{f_l} \cdot f_r) \\ &= ITE(f_l, \overline{f_r}, f_r). \end{aligned}$$

By computing $ITE(f_l, \overline{f_r}, f_r)$ without creating a new $\oplus$-node, the size of the resulting graph is bounded by $O(|P_{f_l}| \cdot |P_{f_r}|)$. Thus, in theory we might achieve up to a quadratic gain in size for each introduced $\oplus$-node. Of course, this result must not be generalized, since, $ITE(f_l, \overline{f_r}, f_r)$ might be much smaller than the given bound, because the sharing of subgraphs, and it also holds only for functions, which are containing $\oplus$-operations. The $\oplus$-operator alone does not form a complete Boolean basis and therefore, not all Boolean operators can be mapped to $\oplus$.
For symbolic simulation with $\oplus$-OBDDs, circuits that are containing $\oplus$-/$\equiv$-gates automatically provide a certain number of $\oplus$-operator nodes to be included into the $\oplus$-OBDD. But, there might also be implicit $\oplus$-/$\equiv$-gates hidden in the given circuit description. The simple equivalence

$$f_1 \oplus f_2 = (f_1 \cdot \overline{f_2}) + (\overline{f_1} \cdot f_2)$$

is encoding a $\oplus$-function in an $\{\wedge, \vee\}$-based circuit, which can also be translated into a $\oplus$-node. For CMOS implementations most circuits are based on NAND or NOR gates [McC86]. Here, we can also look for NAND/NOR based encodings of $\oplus$-/$\equiv$-operations with the help of the following equivalences. First, we show how to express $\equiv$ by using NOR gates only. For readability we start from $a \equiv b$, deriving a NOR-based expression:

$$\begin{aligned} a \equiv b &= (a \cdot b) + (\overline{a} \cdot \overline{b}) \\ &= (a \cdot b) + (\overline{a + b}) \\ &= \overline{\overline{(a + (\overline{a + b})) \cdot (b + (\overline{a + b}))}} \\ &= \overline{(a + (\overline{a + b})) + (b + (\overline{a + b}))} \end{aligned}$$

On the other hand, we can express $\oplus$ by only using NAND gates:

$$\begin{aligned} a \oplus b &= (a + b) \cdot (\overline{a} + \overline{b}) \\ &= (a + b) \cdot (\overline{a \cdot b}) \\ &= \overline{(a \cdot (\overline{a \cdot b})) + (b \cdot (\overline{a \cdot b}))} \\ &= \overline{((\overline{(a \cdot (\overline{a \cdot b}))}) \cdot (\overline{(b \cdot (\overline{a \cdot b}))}))} \end{aligned}$$

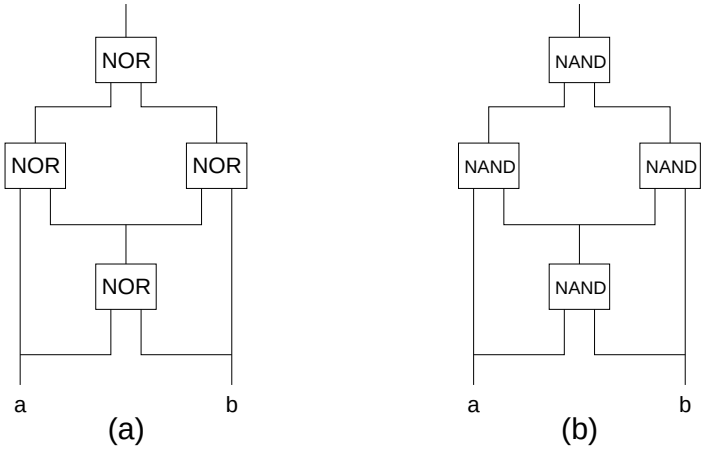


Figure 5.1: NAND and NOR Realizations of $\oplus$ (a) and $\equiv$ (b).

(See Fig 5.1 for an illustrating example).
This encoding scheme really occurs in the circuits of the benchmark set that we have analyzed for our experimental work. E.g., the two combinatorial circuits $C499$ and $C1355$ are functionally equivalent, while $C499$ is utilizing $\oplus$-gates and $C1355$ is utilizing the given NAND encoding scheme.
Now that we have given a first theoretical bound on $\oplus$-node impact on $\oplus$-OBDD size and possible encoding schemes for the $\oplus$-operator, we focus on the practical impact of $\oplus$-nodes in symbolic simulation of the circuits that are contained in our benchmark set.

5.1.2 Experimental Setup

For an estimation of the influence of the number of $\oplus$-nodes in an arbitrary $\oplus$-OBDD we consider the following experimental setup. If we are working with the $\oplus$-OBDD package while using the extended apply-$\oplus$ algorithm exclusively, in most cases too many $\oplus$-nodes are created and thus, affecting $\oplus$-OBDD size in a negative way. To lower this effect, we merge the application of the ITE-$\oplus$ algorithm and extended apply-$\oplus$, while limiting the usage of extended apply-$\oplus$ to a fixed percentage. This approach gives way to include a fixed percentage of additional $\oplus$-nodes into the $\oplus$-OBDD.
Now, for finding out what is the suitable number of $\oplus$-nodes for a given circuit we conduct symbolic simulation of our benchmark set with the following ratio of extended apply-$\oplus$ usage: 50%, 20%, 10%, 5%, and 2.5%.
To achieve a more reliable result about the effects of $\oplus$-node frequency, the extended apply-$\oplus$ algorithm is executed several times at random, but with the fixed rate given above. For every single benchmark circuit and for every given proportion we perform 10 single test runs. Thus, for evaluating the results of the experiments we have computed minimum, maximum, and average number of nodes per circuit and per given percentage over all simulation runs.
In this way we have the possibility to analyze the influence of $\oplus$-node frequency on $\oplus$-OBDD size, and also to gain some insight into the effect of $\oplus$-node place-

ment by analyzing the deviation from the achieved average result.

5.1.3 Experimental Results

In Tables 5.1 and 5.2 we have listed the significant results of the described experimental setup. For a complete overview of all test results, see Table A.3 in the Appendix. The first column of the tables denotes the circuit's name, while the second column for a comparison lists the size of a regular OBDD for this circuit. The next five columns register the $\oplus$-OBDD sizes for the circuit under consideration with a ratio of 50%, 20%, 10%, 5%, and 2.5% of extended apply-$\oplus$ usage, each. For every circuit there are 3 rows listed in the table. The first row lists the minimal $\oplus$-OBDD size that could be achieved within the 10 test runs. In the second row we have computed the arithmetic average $\oplus$-OBDD size achieved in the test runs, and in the last row the maximum $\oplus$-OBDD size is listed. The very last row in Table 5.2 contains the overall sum of the sizes of the data structure for each group of experiments, also minimum, average, and maximum, each. In Tables A.3-A.7 in the Appendix, for each circuit an additional row, containing the standard deviation of the overall results of that particular circuit in relation to the arithmetic average is given as a percentage. For readability, the minimum achieved size for each circuit is denoted in bold face.
For the analysis of the experiments, let us first consider the different aspects that are to be investigated:

(1) What is the influence of a different $\oplus$-node ratio on the $\oplus$-OBDD size in general?

(2) For a fixed $\oplus$-node ratio, given that the $\oplus$-nodes are placed at random within the $\oplus$-OBDD, are there significant differences in size? Leading us to the question, what is the influence of $\oplus$-node placement on the $\oplus$-OBDD size?

Considering these aspects, in the Tables 5.1 and 5.2 we have selected 22 circuits out of our benchmark set of 61 circuits that were able to finish computation under the given resource limitations, for which this experiment was conducted.

ad (1): Considering the $\oplus$-node ratio within a $\oplus$-OBDD, we can deduct the two following opposite peculiarities: As, e.g. the circuits $s510$, $s420$, or mux, there are circuits that seem to benefit of including a large fraction of $\oplus$-nodes. The more $\oplus$-nodes are part of the $\oplus$-OBDD, the smaller they become in size. In general, these circuits do always benefit from the introduction of $\oplus$-nodes.

On the other hand, there are also circuits as, e.g. $i9$, my_adder, or $mult16a$, where the opposite holds. The less $\oplus$-OBDDs are included into the $\oplus$-OBDD, the smaller they become in size. For $i9$ or $bigkey$ the original OBDD size without any $\oplus$-nodes is the smallest. These circuits in general seem not to benefit from the introduction of $\oplus$-nodes in any sense.

Circuit	OBDD - size		⊕-OBDD size				
			50%	20%	10%	5%	2,5%
sbc	3715	min	4302	3658	3562	**3531**	3700
		avg	4588	3987	3861	3806	3793
		max	5038	4219	4274	3961	3993
s635	656	min	**655**	**655**	656	**655**	656
		avg	659	656	656	655	656
		max	663	659	657	656	659
s510	19076	min	**704**	712	19016	19020	19076
		avg	6272	17251	19077	19071	19078
		max	19198	19119	19149	19095	19085
s420	262227	min	**495**	139219	131472	131474	254754
		avg	107602	236890	210778	249153	261480
		max	254760	262244	262239	262236	262233
s208	1033	min	**140**	624	623	624	1031
		avg	610	993	914	992	1033
		max	1043	1040	1039	1034	1038
s1423	98454	min	**93909**	**93909**	97702	98557	97836
		avg	104907	100641	101243	100175	98919
		max	116530	106276	107074	104556	100470
dsip	13921	min	**13175**	13487	13676	13832	13873
		avg	13432	13683	13795	13869	13910
		max	13733	13858	13902	13933	13971
x3	2760	min	1804	**1526**	2665	2789	2604
		avg	2799	2755	2836	2852	2775
		max	3365	3022	3020	2991	2862
my_adder	327677	min	297773	262222	262190	262192	**262188**
		avg	390205	310440	285503	292756	264034
		max	465113	369380	361218	348200	266310
mux	131071	min	**217**	131071	131071	131071	131071
		avg	91814	131071	131071	131071	131071
		max	131071	131071	131071	131071	131071
i9	**2278**	min	5263	3431	2789	2506	2380
		avg	5757	3717	2992	2633	2451
		max	6215	3962	3367	2790	2564
example2	469	min	479	467	**456**	466	468
		avg	499	483	475	475	472
		max	516	496	485	483	482
cm150a	131071	min	**207**	**207**	**207**	131071	131071
		avg	39466	117984	117984	131071	131071
		max	131071	131071	131071	131071	131071
booth8x8	6386	min	7486	6636	6120	5975	**5964**
		avg	10410	8489	6851	6380	6404
		max	12550	10752	9040	7252	7648
apex1	28336	min	**15504**	24051	21372	21508	26536
		avg	23346	28374	26521	27338	28095
		max	28972	30908	30095	28740	28728

Table 5.1: Influence of ⊕-Node Frequency on ⊕-OBDD Size (Part 1)

Circuit	OBDD - size		⊕-OBDD size				
			50%	20%	10%	5%	2,5%
alu32r	189266	min	**63974**	96157	153622	170621	162704
		avg	92939	146516	167093	182239	182294
		max	136123	174227	188298	189982	190099
alu32	12194	min	**5192**	8003	10096	10799	11679
		avg	6877	9839	11022	11572	12004
		max	8403	10831	11886	12211	12321
adder16	327812	min	320025	285085	262411	**262310**	**262310**
		avg	420763	342228	287074	275827	264098
		max	537425	455693	398660	356718	273939
C499	45922	min	7425	7150	7036	**7030**	**7030**
		avg	10190	7683	7846	7100	7080
		max	13254	8652	13308	7260	7272
bigkey	**6170**	min	8051	7157	6639	6417	6253
		avg	8139	7310	6770	6479	6322
		max	8225	7426	6862	6555	6369
rot	166674	min	202102	170609	170856	167104	**163410**
		avg	217670	188105	178755	172378	167767
		max	235279	215030	189352	177577	175387
mult16a	360442	min	408863	279404	265291	262190	**262188**
		avg	509929	330139	309691	311544	280296
		max	605804	376504	451340	451611	328478
Σ	2.406.120	min	1.723.875	1.810.664	1.842.396	2.003.041	2.125.959
		avg	2.386.097	2.326.164	2.200748	2.254.635	2.186.692
		max	3.114.133	2.686.855	2.678.278	2.589.747	2.278.346

Table 5.2: Influence of ⊕-Node Frequency on ⊕-OBDD Size (Part 2)

ad (2): For each circuit and for each given ratio we conducted 10 independent experiments, for evaluating the achieved node size for $\oplus$-OBDDs with a random placement of $\oplus$-nodes at a fixed ratio. For determining the significance of the different $\oplus$-node placement in each group, we have computed the standard deviation

$$\sigma = \sqrt{\frac{1}{n-1} \cdot \sum_{i=1}^{n} (x_i - \overline{x})^2}$$

of the 10 experiments from the computed average value. n denotes the number of experiments, x_i the node size achieved in experiment i, $1 \leq i \leq n$, and $\overline{x}$ is the arithmetic average $\overline{x} = \frac{1}{n} \cdot \sum_{i=1}^{n} x_i$.

The deviation for each circuit and each fixed ratio is included in the Tables in the Appendix.

In general, circuits that benefit from the introduction of $\oplus$-nodes do have a larger deviation in size for $ratio = 50\%$, compared to the deviation of the circuits belonging to the opposite group. For smaller $ratio < 50\%$ of $\oplus$-nodes, also the deviation becomes smaller. A very significant example is $s420$, where the minimum achieved size is 495 nodes, while with the same *ratio* the worst achieved result computes to 254760 nodes. A similar behavior can be found for *mux* or $cm150a$.

Considering the overall sum of nodes, the results for the minimum achieved sizes range from 71% to 88% of the original OBDD size, where larger *ratio* is producing smaller results. The opposite holds for the sum of the average and the maximum results. There, with a smaller *ratio*, the achieved sizes become smaller, meaning that the possible worse effects of $\oplus$-node introduction are balanced, when *ratio* is becoming small. Nevertheless, the average sizes are ranging from 44% to 91%, while the maximum sizes are ranging from 129% to even 95%. This signifies that even if we are only considering the worst case behavior in the experiment, we are able to gain some profit against regular OBDDs.
Secluding our analysis we are able to state the following results:

- There are circuits that do seriously benefit from the introduction of $\oplus$-nodes in general.
- For most of these circuits it holds that many $\oplus$-nodes within the $\oplus$-OBDD result in a smaller overall $\oplus$-OBDD node size.
- In the average, the smallest overall $\oplus$-OBDD sizes could be achieved by introducing only a small percentage of $\oplus$-nodes into the $\oplus$-OBDD.
- Due to our results summarized in (2), we can conclude that the placement of $\oplus$-nodes also seriously effects the resulting $\oplus$-OBDD size. Thus, motivating further research on that particular subject.

5.2 $\oplus$-Node Placement

As we could see in the previous experiments, not only the number of $\oplus$-nodes deployed in a $\oplus$-OBDD, but also their placement is an important factor that is severely affecting the final size of the data structure. But, how can we find out about the best suited placement for $\oplus$-nodes, and can we possibly change the position of a $\oplus$-node in an already constructed diagram?
When dealing with symbolic simulation, the first thing is to reflect on the given circuit topology. There, every $\oplus$-/$\equiv$-gate or every gate combination that is identified to represent a $\oplus$-/$\equiv$-operation can be directly translated into a $\oplus$-node. From the placement inside the given circuit description we might conclude that introducing $\oplus$-nodes right at the point where they are occurring in the circuit netlist should be an appropriate place. But, this approach is only working for circuits that really do contain $\oplus$-/$\equiv$-operators. For circuits, which don't comprise that operation, it is much more difficult to find an appropriate place for inserting a $\oplus$-node. Also for other tasks like sequential verification or other optimization problems that can be solved with the help of $\oplus$-OBDDs it is difficult to find a qualified insertion point.

5.2.1 A Simple $\oplus$-Node Placement Heuristic

As an indicator, whether the introduction of a $\oplus$-node at a specific place into the $\oplus$-OBDD might be useful or not, we might consider the following assumption: If the introduction of a $\oplus$-node is useful, then its introduction must result in a $\oplus$-OBDD of smaller size. Now, we might place $\oplus$-nodes randomly into the already constructed $\oplus$-OBDD and decide, whether to keep them or not according to their effect on the $\oplus$-OBDD size. But, this approach requires the construction of the complete $\oplus$-OBDD first, before we have the possibility to start the improvement of its size. Thus, we could think of constructing only a part of the $\oplus$-OBDD, introducing a satisfactory number of $\oplus$-nodes, and afterwards, continue with its construction.
By following this concept, we end up in a dynamic approach, which in each construction step of the $\oplus$-OBDD compares its size with and without introduced $\oplus$-node at the place under current consideration. In symbolic simulation of a combinatorial design this means, for each single gate G, we construct the complete $\oplus$-OBDD P_G representing the function f_G of G. First, we use the ITE-$\oplus$ algorithm and construct P_{f-ite}, and additionally, we perform the same computation with the pDE(nDE)-Apply algorithm, resulting in P_{f-nDE} (P_{f-pDE}). Next, we compare the two distinct $\oplus$-OBDD sizes and decide, which $\oplus$-OBDD to keep. If $|P_{f-ite}| > |P_{f-nDE}|$ ($|P_{f-pDE}|$), then, we keep P_{f-nDE} (P_{f-pDE}) and vice versa.
Thus, locally we always try to make use of the smallest possible $\oplus$-OBDD. But, of course this is only a local minimum. Another disadvantage is that we always have to construct both versions of the $\oplus$-OBDD with the two synthesis procedures. To increase the efficiency of the approach, we limit the construction of the alternative $\oplus$-OBDDs to the case, only when the regular ITE-$\oplus$ algorithm computes a $\oplus$-OBDD that is passing a certain fixed threshold. Thus, the ad-

```
Input: ⊕-OBDD P_f, P_g, and operator ⊗
Output: ⊕-OBDD P_res, representing res = f ⊗ g

local_greedy_synthesis(P_f, P_g, op) {
  res-ite = ITE-PLUS(P_f, P_g, ⊗);
  if ( size(res-ite) > threshold ) {
    res-alt = APPLY-⊕(P_f, P_g, ⊗);
    if ( res-alt < res-ite ) {
      res = res-alt;
      delete res-ite;
    } else {
      res = res-ite;
      delete res-alt;
    }
  }
  return(res);
}
```

Figure 5.2: Locally Greedy Heuristic for ⊕-OBDD Optimization.

ditional construction of ⊕-OBDDs is limited to the cases, when an alternative representation will be of major advantage. For an outline of this locally greedy algorithm see Fig. 5.2.
An important factor for the efficiency of this heuristic is the proper choice of the threshold value. For the conducted experiments, we have chosen from the following possibilities:

- Forget about the threshold value, construct the ⊕-OBDD both for ITE-⊕ and APPLY-⊕, compare their sizes and choose the smaller one.
- Set the threshold value to the maximum size of the two operands multiplied by a constant c, thus $t = c \cdot \max(|P_f|, |P_g|)$ (MAX).
- Set the threshold value to the sum of the sizes of the two operands multiplied by a constant c, thus $t = c \cdot (|P_f| + |P_g|)$ (ADD).

In Table 5.4 the results of these experiments are put together. For different constant factors $0.6 \leq c \leq 2.0$ we have listed the overall size achieved for all benchmarks for the methods denoted as MAX, with application of nDE/pDE, and as ADD, as referred in the list above. Note that compared to the standard synthesis of the previous chapter, only 63 out of the 67 circuits of the benchmark set were able to finish the computation within the given resource limitations Additionally, we have tried to reverse the decision criteria for the heuristic, i.e. we use Apply-⊕ as default function decomposition and only in the case, when the given threshold value is exceeded by the resulting ⊕-OBDD, we switch to the ITE-⊕ algorithm with the regular Boole/Shannon-decomposition. For this strategy (further denoted as *pDE-first/nDE-first*), where much more

OBDD-size		⊕-OBDD size			
OBDD	%	pDE	%	nDE	%
4.468.873	100	3.261.714	73	4.4687.023	104.9

Table 5.3: Reference Table for OBDDs and ⊕-OBDDs with pDE/nDE.

	⊕-OBDD size					
c	**MAX (pDE)**	**%**	**MAX (nDE)**	**%**	**ADD**	**%**
0.6	3.634.456	81.3	3.216.892	72.0	3.214.424	71.9
0.7	3.633.394	81.3	3.215.694	72.0	3.212.698	71.9
0.8	3.469.411	77.6	3.212.683	71.9	3.218.253	72.0
1.0	3.213.905	71.9	**2.997.931**	67.1	**3.008.490**	67.3
1.2	**3.000.038**	67.1	2.999.501	67.2	3.009.105	67.3
1.5	3.000.179	67.1	3.009.641	67.3	3.011.906	67.4
2.0	3.013.619	67.4	3.015.341	67.5	3.016.222	67.5
c	**MAX (pDE-first)**	**%**	**MAX (nDE-first)**	**%**	**ADD**	**%**
1.0	4.029.754	90.2	4.202.453	94.0	4.208.246	94.2

Table 5.4: Locally Greedy Heuristic for ⊕-Node Placement.

⊕-nodes are created, in general the achieved overall sizes are worse compared to the original approach and only the results for the best choice of the threshold parameter $c = 1.0$ are listed for that case. This fact confirms the previously achieved result that a small number of ⊕-nodes placed at well chosen positions inside the ⊕-OBDD provides the best overall effect in the average. For a reference in Table 5.3 the overall sizes for OBDDs and for exclusive application of pDE and nDE are also listed. The values given in percentages are always referring to the OBDD size, which is denoted as 100%. For ADD we have only listed the achieved sizes for nDE, because for pDE the results are only slightly different. For a complete overview of the achieved results see Tables A.8 to A.11 in the Appendix.

By comparing the overall achieved size, the first thing to mention is that the exclusive application of pDE results in 27% gain in size, compared to a 5% loss for nDE, if related to the original OBDD size. By applying the locally greedy heuristic with different constant parameter $0.6 \leq c \leq 2.0$, the ⊕-OBDD size becomes minimal for choosing the parameter $c \approx 1.0$. There, we are able to achieve an up to 33% win for the overall size, which is much better compared to the exclusive application of nDE or pDE. In their general behavior the two approaches MAX and ADD produce only slight differences in size.

For circuits that benefit from the introduction of ⊕-nodes, the exclusive application of nDE/pDE is often better than the proposed heuristics. But, for circuits that don't benefit from the introduction of ⊕-nodes, the heuristic is of-

	Overal Runtime					
	OBDD	$\oplus$**-OBDD**				
		pDE	**MAX(pDE)** $c = 1.2$	**nDE**	**MAX(nDE)** $c = 1.0$	**ADD(nDE)** $c = 1.0$
Σ	86.57	2562	2528	2961	3099	3160
normalized	**1**	29.6	29.2	34.2	35.8	36.5

Table 5.5: Overall Time Requirement for Locally Greedy Heuristic.

ten producing much better results than the exclusive application of pDE/nDE. For the circuit *mult*16*a*, e.g the heuristic is always producing a smaller result compared to OBDD size or the exclusive application of nDE/pDE. But, on the other hand, for *cm*150*a*, the heuristic is not able to reproduce a result of similar quality compared to the exclusive application of pDE/nDE. For all methods and all parameters the achieved $\oplus$-OBDD size is approximately of the size of the OBDD or better.
Thus, in general our heuristic is able to keep the benefits of both decompositions, BS and nDE/pDE.
For the locally greedy heuristic the runtime still is not comparable to the OBDD performance of the CUDD package. But, if we compare the runtime of synthesis with and without the heuristic, if the heuristic is applied, the runtime increases in the average of less than 10% (see Table 5.5). For the first method *MAX(pDE)*, the runtime of the synthesis algorithm together with the additional heuristic is even faster, which is due to the fact that in that case in the average the $\oplus$-OBDDs are about 50% smaller than without the heuristic applied. Considering the general reduction in size, spending a little overhead of time is certainly worth while.

5.2.2 Using Linear Combinations

Another – maybe a more sophisticated – way of determining an appropriate position for introducing a $\oplus$-node can be derived from linear algebra.
It is convenient to regard the space $\mathbb{B}_n = \{f | f : \{0,1\}^n \to \{0,1\}\}$ of Boolean functions of n variables as an algebra over the two-element field $\mathbb{Z}_2$, i.e. a 2^n-dimensional vector space with an additional multiplication operation. The product of $f, g \in \mathbb{B}_n$, which corresponds to coordinate-wise conjunction, is denoted by $f \cdot g$ and the sum, which corresponds to coordinate wise XOR, by $f+g$. In this context, the variable x_i is taken to represent the projection from $\{0,1\}^n$ to the ith coordinate and $\overline{x_i}$ as the according complement.
Now, let P be a $\oplus$-OBDD representing $f_P \in \mathbb{B}_n$. The Boolean function f_P can be regarded as the Boolean function assigned to the top node v of P and can be defined inductively on $i = n+1, n, \ldots, 0$, where i denotes the level of P to which the node v belongs:

(i) $i = n+1$: v is sink, $f_v = 1/0$.

(ii) v is a node in level i, $1 \leq i \leq n$: let f_v^1 be the function computed by v's 1-successor and f_ν^0 the function computed by its 0-successor, respectively. Then $f_v = x_i \cdot f_v^1 + \overline{x_i} \cdot f_v^0$.

If it is possible to express the function under consideration as a linear combination of already computed functions,

$$f = \sum_{i=1}^{m} f_i, \ m \in \mathbb{N},$$

we are able to represent this function as a tree of $m-1$ $\oplus$-nodes connected to the m $\oplus$-OBDDs representing the already computed functions f_i, $1 \leq i \leq m$. Unfortunately, for testing the possibility of an existing solution containing the functions that have already been computed, all possible linear combinations of $\oplus$-OBDDs represented in the unique table have to be tested and thus, making this approach not viable in that way, because too much time is required.
In the already mentioned model of POBDDs, this concept is utilized by constructing new POBDD nodes out of linear combinations of base vectors of the underlying vector space, where the according linear combination can be derived by the solution of a system of linear equivalences.

5.2.3 Dynamic $\oplus$-Node Placement

Another possible approach is the insertion of additional $\oplus$-nodes into a $\oplus$-OBDD either at random or by some heuristic. Then, after the construction is finished, the $\oplus$-nodes can be moved around to find a better and more suitable positioning for them. For realizing this concept, we have to find out, if it is possible to move a single $\oplus$-node efficiently by exchanging it with an adjacent node. But, for this task, first, we have to consider an efficient way for the implementation of $\oplus$-nodes.
The first question that we have to answer is how to integrate $\oplus$-nodes into the approved implementation of the OBDD data structure. Considering single nodes of a $\oplus$-OBDD we can use the same data structure for $\oplus$-nodes as for regular branching nodes. The only difference to OBDD nodes is that besides labeling a node with a given variable, we have to denote, whether the node is either a branching node or a $\oplus$-node. This can be done by a single spare bit within the integer data field denoting the variable that the node is labeled with, e.g. we use the most significant bit representing the sign of that integer value.
As mentioned in a previous chapter, for OBDDs, branching nodes are stored in different hash tables according to the variable that the node is labeled with. This implementation is chosen to ensure a fast access to all the nodes that are labeled with this specific variable. Fast access to nodes that are labeled with the same variable is rather important for the dynamic exchange of adjacent variables to improve a previously given variable order.
For efficient storage of $\oplus$-nodes also efficiently, we have to consider different possible solutions and to analyze their effects:

(1) Create one additional hash table $H_\oplus$ that is serving as a container for all $\oplus$-nodes in the $\oplus$-OBDD.

As we have already mentioned in a previous section, it is advantageous, if we label each $\oplus$-node with a reference to the variable of the successor branching node that is first w.r.t. the given variable order π. This method implies the following two additional possibilities:

(2) For every variable $x_i \in \{x_1, \ldots, x_n\}$ in addition to the already existing hash table, create another hash table $H_{x_i-\oplus}$, which contains all $\oplus$-nodes that are labeled with a reference to this variable x_i.

(3) Use the already existing hash table H_{x_i} that is containing the branching nodes labeled with x_i and hash all $\oplus$-nodes that are labeled with a reference to x_i also in H_{x_i}.

Approach (1) benefits from the fact that all $\oplus$-nodes are hashed within the same table. Thus making all those accesses faster, where it is necessary to reach all $\oplus$-nodes together as fast as possible. On the other hand, for accessing only those $\oplus$-nodes that are labeled with a reference to a given variable x_i, all $\oplus$-nodes of the $\oplus$-OBDD have to be examined.
To prevent this disadvantage, one of the alternative solutions would be preferable. In approach (2), for every single variable x_i we have to create a new additional hash table that is containing only $\oplus$-nodes with a reference to x_i. Thus, on the one hand giving fast access to all $\oplus$-nodes of the $\oplus$-OBDD and, on the other hand, giving fast access to only those $\oplus$-nodes, which have a reference to a fixed variable x_i.
But, why should we create new hash tables at all? Dynamic reordering operations, where $\oplus$-nodes and branching nodes are exchanged between adjacent tables would become much more complicated, because transitions of nodes to and from the additional hash tables $H_{x_i-\oplus}$ have to be considered. Since, all $\oplus$-nodes contain the reference to a given variable x_i, we can hash these $\oplus$-nodes also within the same hash tables as all other branching nodes that are labeled with the same variable x_i, as it is proposed in approach (3).
There, it is also easier to determine, whether a new node that is to be created does already exist, because only one hash table H_{x_i} has to be accessed for that operation. Because of the missing canonicity for $\oplus$-OBDDs, there might be $\oplus$-nodes $v_{\oplus-f}$ representing the same Boolean function f as being represented by some different branching node v_{b-f}. Before a new branching node v_b that is labeled with the variable x_i can be created, in approach (1) the hash table H_{x_i} and the hash table $H_{\oplus}$ have to be examined, whether a node with the same functionality does already exist. The same holds for the insertion of a new $\oplus$-node that is labeled with a reference to x_i. And also solution (2) necessitates the access to both the two hash tables H_{x_i} and $H_{x_i-\oplus}$ for the decision, whether a new node does already exist. Therefore, approach (3) has been considered to be the most efficient approach and has been preferred for the implementation of our $\oplus$-OBDD package.
If we consider the case that the given $\oplus$-OBDD is completely constructed first and if its size should be improved without changing the given variable order π, we must have the possibility to move $\oplus$-nodes inside the $\oplus$-OBDD data structure up and down. With the help of the following equivalences an

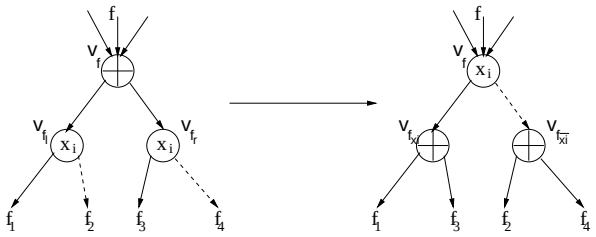


Figure 5.3: Exchange of $\oplus$-Nodes and Branching Nodes.

exchange of a $\oplus$-node with an adjacent branching node can easily be implemented. W.l.o.g. we consider a $\oplus$-OBDD P_f representing the Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ with a $\oplus$-node v_f at the top. v_f is connected to two successing branching nodes v_{f_l} and v_{f_r}, both labeled with the same variable $x_i \in \{x_1, \ldots, x_n\}$. v_{f_l} and v_{f_r} are connected to f_1, f_2, and f_3, f_4, respectively (see Fig. 5.3). Note that according to our implementation all three nodes v_f, v_{f_l}, and v_{f_r} reside in the same hash table H_{x_i}.

$$\begin{aligned}
f &= f_l \oplus f_r \\
&= (x_i \cdot f_{l_{x_i}} + \overline{x_i} \cdot f_{l_{\overline{x_i}}}) \oplus (x_i \cdot f_{r_{x_i}} + \overline{x_i} \cdot f_{r_{\overline{x_i}}}) \\
&= (x_i \cdot f_{l_{x_i}} \oplus \overline{x_i} \cdot f_{l_{\overline{x_i}}}) \oplus (x_i \cdot f_{r_{x_i}} \oplus \overline{x_i} \cdot f_{r_{\overline{x_i}}}) \\
&= (x_i \cdot f_{l_{x_i}} \oplus x_i \cdot f_{r_{x_i}}) \oplus (\overline{x_i} \cdot f_{l_{\overline{x_i}}} \oplus \overline{x_i} \cdot f_{r_{\overline{x_i}}}) \\
&= x_i(f_{l_{x_i}} \oplus f_{r_{x_i}}) + \overline{x_i}(f_{l_{\overline{x_i}}} \oplus f_{r_{\overline{x_i}}})
\end{aligned}$$

Thus, the $\oplus$-node on top of P_f can be moved down by one position without changing v_f's functionality. Note also that the implementation of this node exchange also benefits from the method we have chosen for hash table arrangement concerning the storage of $\oplus$-nodes in the previous section, i.e. node v_f remains in the same hash table and at the same position while changing from a $\oplus$-node into a branching node.
By moving move v_f down by one position in P_f, the following operations take place:

- The node v_f changes its identity into a branching node labeled with x_i. This means that the node v_f does not change its functionality and therefore, its signature. It remains in the same hash table at exactly the same position.

- Then, v_f is connected to two new successors $v_{f_{x_i}}$ and $v_{f_{\overline{x_i}}}$. The node $v_{f_{x_i}}$ is a $\oplus$-node connected to f_1 and f_3, while $v_{f_{\overline{x_i}}}$, which is also a $\oplus$-node, is connected to f_2 and f_4. If $v_{f_{x_i}}$ and $v_{f_{\overline{x_i}}}$ do not already exist in P_f, they have to be newly created.

- The references to the old successors v_{f_l} and v_{f_r} are deleted and their reference counter is decremented by one. If no other node is pointing to v_{f_l} and v_{f_r} anymore, i.e. if their reference counter is decremented to 0, the nodes will be put in a separate list for garbage collection,

See Fig. 5.4 and Fig.5.5 for an outline of the $\oplus$-node exchange algorithm in pseudo code.
In the case of $f_1 = f_3$ we can immediately reduce $v_{f_{x_i}}$ to the 0-sink, because $f_{x_i} = f_1 \oplus f_3 = f_1 \oplus f_1 = 0$. The same holds for $f_2 = f_4$. Then, $v_{f_{x_i}}$ ($v_{f_{\overline{x_i}}}$) does not have to be created, but v_f is directly connected with the 0-sink via its 1-edge (0-edge) (see Fig. 5.6).
When the exchange is taking place in the opposite direction, i.e. the $\oplus$-OBDD P_f under consideration with a branching node v_f at the top labeled with x_i being exchanged with its successing $\oplus$-nodes $v_{f_{x_i}}$ and $v_{f_{\overline{x_i}}}$, then the following reduction is possible: If $f_2 = f_3$, then the right successor v_{f_r} of v_f, now being a $\oplus$-node can directly reduced to f_2. On the other hand, if $f_1 = f_4$, then the left successor v_{f_l} of v_f can be directly reduced to f_1 (see Fig. 5.7).
If complemented edges are used, then we have to take special care only for those edges that are connecting v_f with v_{f_l} and v_{f_r}. The complements on the other edges, i.e. on all edges pointing to v_f and those pointing from v_{f_l} to f_1, f_2, and from v_{f_r} to f_3, f_4 are not affected by the node exchange. So, if the edge from v_f to v_{f_r} is inverted, then this situation can be managed by applying the equivalence rules for inverted edges on branching nodes. Thus, after the node exchange, the edge directed to $v_{f_{\overline{x_i}}}$ is not complemented, while the edges pointing to f_3 and f_4 will be complemented (See Fig. 5.8).
Whenever complemented edges are used, additional possibilities for reductions have to be taken into account. If $f_2 = \overline{f_4}$, then the according edge leaving v_f can be directly connected to the 1-sink, because of the equality $f_{\overline{x_i}} = f_2 \oplus f_4 = f_2 \oplus \overline{f_2} = 1$. The same holds for f_1 and f_3 (See Fig. 5.9).
Because $\oplus$-OBDDs are not canonical, considering uniqueness rules for complemented edges are not useful in the same way as in the case of OBDDs. But, for improving cache efficiency and simplifying the implementation, we can nevertheless avoid the inversion of 1-edges in case of branching nodes, respectively the inversion of *left*-edges for $\oplus$-nodes.
But, the important question is, how does the exchange of $\oplus$-nodes and branching nodes affect the $\oplus$-OBDD size? Let us first take a look on a $\oplus$-OBDD P_f, where we are exchanging a branching node v_f labeled with x_i with its successor $\oplus$-nodes $v_{f_{x_i}}$ and $v_{f_{\overline{x_i}}}$ (see Fig. 5.10 from left to right), resulting in a $\oplus$-OBDD P'_f. In the worst case, for each branching node in level i two additional new branching nodes v_{f_l} and v_{f_r} have to be created. For simplicity, whenever we refer to the exchange operation for $\oplus$-nodes and branching nodes, in the case when a branching node is to be exchanged with its $\oplus$-node successors, we will denote the exchange as the *swap-up* operation.
Thus, in the worst case for the swap-down operation, the size of P'_f is bounded by $2 \cdot |P_f|$. In the best case, if w.l.o.g. $f_1 = f_3$, then for each branching node in level i one new branching node might be created, but it may already exist in level i, while the old successor nodes $v_{f_{x_i}}$ and $v_{f_{\overline{x_i}}}$ become obsolete, because no other node does have a reference to them. Thus, in the best case, the size of P'_f must be greater or equal than $\frac{|P_f|}{2}$ and we can conclude the following theorem:

Theorem 5.1 *Exchanging all branching nodes of a given variable x_i in a $\oplus$-*

```
Input: ⊕-OBDD P_f with top node v_f
Output:⊕-OBDD P_f with top node and successor nodes being exchanged

swap(P_f) { //(a) v_f is ⊕-node
  if (v_f is ⊕-node) {
    label = min(l(v_{f_l}, v_{f_r}));
    if (l(v_{f_l}) == label) {
      f_1 = v_{f_l}.1-succ; f_2 = v_{f_l}.0-succ;
    } else {
      f_1 = f_2 = v_{f_l};
    }
    if (l(v_{f_r}) == label) {
      f_3 = v_{f_r}.1-succ; f_4 = v_{f_r}.0-succ;
    } else {
      f_3 = f_4 = v_{f_r};
    }
    l(v_f) = label;
    if (f_1 == f_3) {
      v_f.1-succ = 0;
    } else if (f_1 == ¬f_3) {
      v_f.1-succ = 1;
    } else {
      v_f.1-succ = find_or_create_new(⊕, f_1, f_3);
    }
    if (f_2 == f_4) {
      v_f.0-succ = 0;
    } else if (f_2 == ¬f_4) {
      v_f.0-succ = 1;
    } else {
      v_f.0-succ = find_or_create_new(⊕, f_2, f_4);
    }
    decrement refcount of v_{f_l} and v_{f_r};
  }
```

Figure 5.4: Algorithm for Exchange of ⊕-Nodes and Branching Nodes (part 1).

```
  else { //(b) v_f is branching node labeled with x_i
    if (l(v_f.1-succ) == ⊕) {
      f_1 = f_{x_i}.1-succ; f_2 = f_{x_i}.0-succ;
    } else {
      f_1 = f_2 = f_{x_i};
    }
    if (l(v_f.0-succ) == ⊕) {
      f_3 = f_{x̄_i}.1-succ; f_4 = f_{x̄_i}.0-succ;
    } else {
      f_3 = f_4 = f_{x̄_i};
    }
    if (f_1 == f_3) {
      v_f.1-succ = f_1;
    } else {
      v_f.1-succ = find_or_create_new(x_i, f_1, f_3);
    }
    if (f_2 == f_4) {
      v_f.0-succ = f_2;
    } else {
      v_f.0-succ = find_or_create_new(x_i, f_2, f_4);      }
    l(v_f) = ⊕;
    decrement refcount of f_{x_i} and f_{x̄_i};    }
  return(v_f);
}
```

Figure 5.5: Algorithm for Exchange of $\oplus$-Nodes and Branching Nodes (part2).

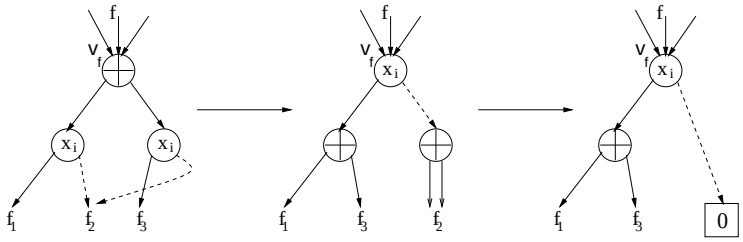


Figure 5.6: Reduction while Exchanging $\oplus$-Nodes and Branching Nodes (1).

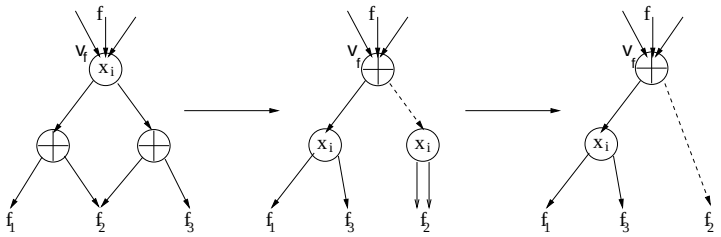


Figure 5.7: Reduction while Exchanging $\oplus$-Nodes and Branching Nodes (2).

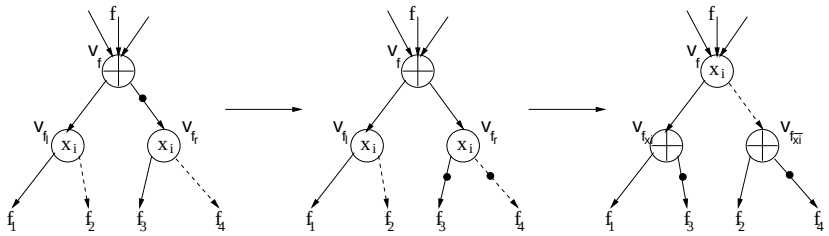


Figure 5.8: Exchanging ⊕-Nodes with Complemented Edges.

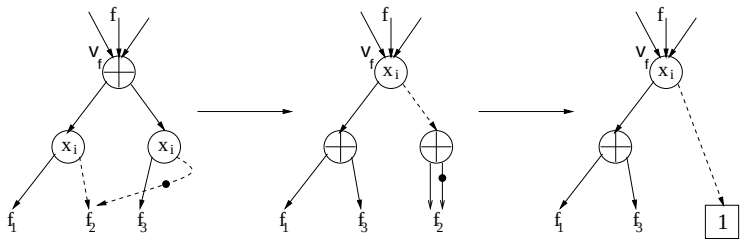


Figure 5.9: Reduction for Exchanging ⊕-Nodes with Complemented Edges.

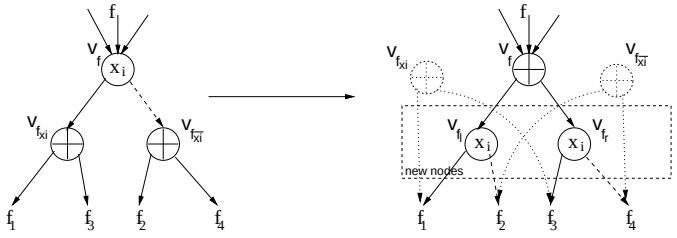


Figure 5.10: Swap-Up Operation of ⊕-Node.

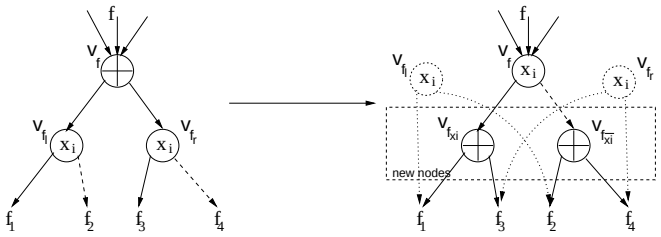


Figure 5.11: Swap Down Operation of $\oplus$-Node.

OBDD P_f with their successing $\oplus$-nodes, i.e. performing a swap-up operation for all branching nodes labeled with x_i, results in a $\oplus$-OBDD P'_f, with size bounded by

$$\frac{|P_f|}{2} \leq |P'_f| \leq 2 \cdot |P_f|.$$

For the case of the exchange of $\oplus$-nodes with their successing branching nodes that are located in the same hash table, different bounds will hold. To distinguish this case from the former situation, the operation of exchanging a branching node with its successing $\oplus$-nodes, will be denoted as *swap-down* operation (see Fig.5.11). Note that it is only suitable to perform a swap-down operation between nodes that are located in the same hash table. The $\oplus$-node is labeled with the variable of its first successor w.r.t. the given variable order. An exchange of that $\oplus$-node with a branching node that is labeled with a different variable is not directly possible, because this also necessitates a change of the variable order and thus, also structural changes for the rest of the $\oplus$-OBDD are required.
While the upper bound for the swap-down operation is the same as for the swap-up operation, the lower bound changes.
For the case that $f_2 = f_4$ ($f_1 = f_3$ can be treated in the same way), the new 0-successor $f_{f_{\overline{x_i}}}$ of v_f computes $f_{\overline{x_i}} = f_2 \oplus f_4 = 0$ and thus, the entire subgraph rooted by $v_{f_{\overline{x_i}}}$ turns to the 0-sink (if $f_2 = \overline{f_4}$, then $v_{f_{\overline{x_i}}}$ will be the 1-sink, respectively). See Fig.5.12 for an illustration of this effect.
If no other node is referencing f_2, this reduction might affect the size of $P_{f'}$ dramatically in the sense that up to $\frac{2^{n-(i+1)}}{n-(i+1)}$ [BHR95], i.e. a complete sub-OBDD rooted by f_2 will become superfluous in the best case for a single swap-down operation. The other successor branch of v_f cannot simultaneously collapse, because otherwise, f would have been previously recognized as being the 0-sink for $f_1 = f_3 \ \wedge \ f_2 = f_4$ (1-sink for $f_1 = \overline{f_3} \ \wedge f_2 = \overline{f_4}$) or the projection function of x_i for $f_1 = \overline{f_3} \ \wedge \ f_2 = f_4$ (or its complement for $f_1 = f_3 \ \wedge \ f_2 = \overline{f_4}$).
Thus, it is difficult to give a lower bound, for exchanging all $\oplus$-nodes labeled with a given variable x_i, because the collapse of one successor of all new branching nodes labeled with x_i has to be considered. Also, for this reason the effects of the swap-down operation are not local anymore and affect the nodes beyond that level.

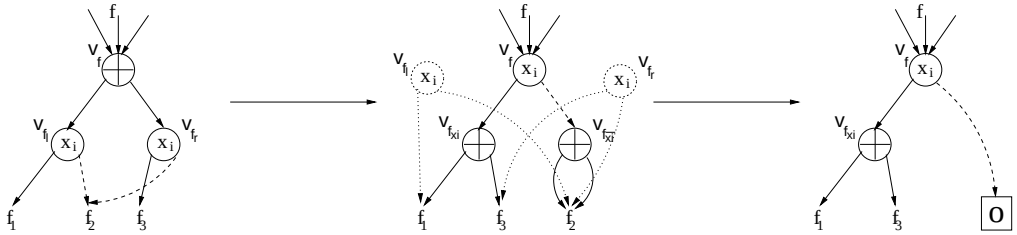


Figure 5.12: Reduction in Swap Down Operation.

Theorem 5.2 *Exchanging all $\oplus$-nodes of a given variable x_i in a $\oplus$-OBDD P_f with their successing branching-nodes, i.e. performing a swap-down operation for all $\oplus$-nodes labeled with x_i, results in a $\oplus$-OBDD P'_f, with size bounded by*

$$MIN_{\oplus-OBDD}(f) \leq |P'_f| \leq 2 \cdot |P_f|,$$

where $MIN_{\oplus-OBDD}(f)$ denotes the minimal $\oplus$-OBDD representation size for the function f for a given variable order.

But, this property also shows the power of the $\oplus$-operation and the importance to find an appropriate place, where to position the $\oplus$-nodes within the $\oplus$-OBDD. We also see that this property can cause an assymetric behavior that has to be further considered for optimization heuristics in the upcoming sections.

5.2.4 Meta-$\oplus$-Nodes

If, for further improvement of the $\oplus$-OBDD size, we want dynamically adjust the given variable order π, we are facing the problem that between two adjacent variables that are to be exchanged, one or even more $\oplus$-nodes might be positioned. Before an exchange of two adjacent variables can be performed, all $\oplus$-nodes between these variables have to be removed by exchanging them with neighbored branching nodes. If more than only a single $\oplus$-node is positioned between two branching nodes that are labeled with adjacent variables, removing these $\oplus$-nodes might lead to a significant blow up in size. Also, all these $\oplus$-nodes might be placed in different hash tables, what might also be a source of new ambiguity.
The situation is much simpler, if only one single $\oplus$-node is located between two branching nodes that are to be exchanged. Then the implementation of the variable exchange operation will become feasible. One possibility to ensure that there is at most one $\oplus$-node between any two branching nodes that are labeled with adjacent variables is to combine all $\oplus$-nodes between these two branching nodes to a single meta-$\oplus$-node that serves as a container for all successor branching nodes of these binary $\oplus$-nodes (see Fig. 5.13).

Definition 5.1 *Let $k \in \mathbb{N}$. A meta-$\oplus$-node is a $\oplus$-node within a $\oplus$-OBDD with an arbitrary number of successors. In a given $\oplus$-OBDD all $\oplus$-nodes $v_{\oplus_1}, \ldots, v_{\oplus_k}$ on a path between two branching nodes labeled with adjacent variables can be*

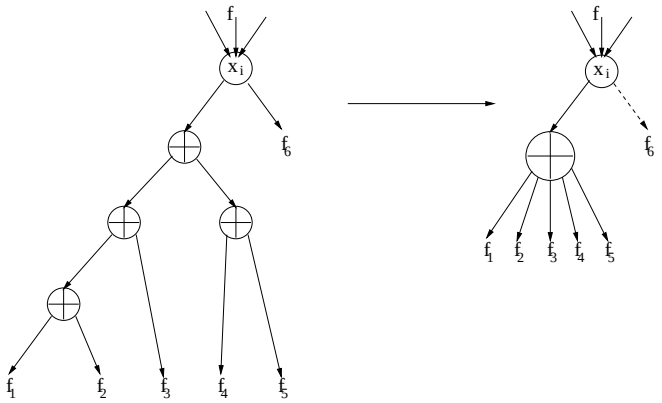


Figure 5.13: Combining Single $\oplus$-Nodes to a Meta-$\oplus$-Node.

summarized into one meta-$\oplus$-node $v_\oplus$. The $k+1$ branching node successors of $v_{\oplus_1}, \ldots, v_{\oplus_k}$ will be the direct successors of $v_\oplus$.

How can meta-$\oplus$-nodes be implemented efficiently? Up to now, in our implementation branching nodes and binary $\oplus$-nodes are differing only by a single bit. But, for the implementation of meta-$\oplus$-nodes, an arbitrary number of successors has to be stored within the meta-$\oplus$-node. One possibility to do this is to maintain the old data structure as a wrapper and thus, it is possible to work with binary $\oplus$-nodes and meta-$\oplus$-nodes simultaneously without introducing a new data structure and thus, keeping the $\oplus$-OBDD implementation homogeneous. The only thing that is changed for meta-$\oplus$-nodes are the two pointers to the left and right successor node. For representing meta-$\oplus$-nodes, we mark the node with a spare bit in the data structure and use the pointer to the right successor for storing the number of successing nodes. The left successor will be a pointer to a successor array that is allocated separately, comprising only as much space as necessary and containing pointers to each single successor (see Fig. 5.14).
One of the advantage of using meta-$\oplus$-nodes is the possibility to apply more sophisticated reduction rules. If we consider a tree of $n \in \mathbb{N}$ binary $\oplus$-nodes the $n+1$ potential successing branching nodes representing $f_1, \ldots, f_{n+1}$ are all connected by a XOR operation, $f = \bigoplus_{i=1}^{n+1} f_i$. Thus, if $\exists i, k \in \{1, \ldots, n+1\}$: $f_i = f_k$ or $f_i = \overline{f_k}$, and f_i, f_k are not necessarily connected to the same binary $\oplus$-node, this possible point of reduction is rather costly to detect in our standard implementation with binary $\oplus$-nodes, because the complete $\oplus$-node subtree has to be accessed in a recursive way. For meta-$\oplus$-nodes we have the possibility to order all successors in the pointer array during insertion time. Thus, we only have to scan the ordered pointer array and can easily detect possible reduction points without stepping through a tree of binary $\oplus$-nodes in depth-first-search (dfs) manner.
If w.l.o.g. $f_n = f_{n+1}$, then

$$f = \bigoplus_{i=1}^{n+1} f_i = \bigoplus_{i=1}^{n-1} f_i \ \oplus 0 = \bigoplus_{i=1}^{n-1} f_i.$$

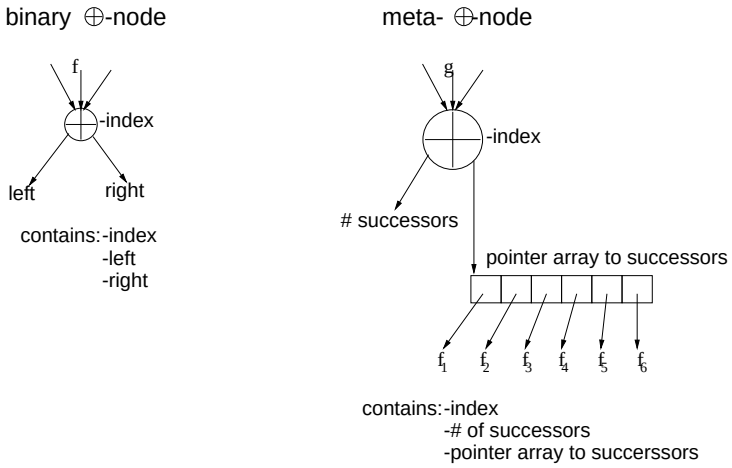


Figure 5.14: Implementation of Meta-⊕-Nodes.

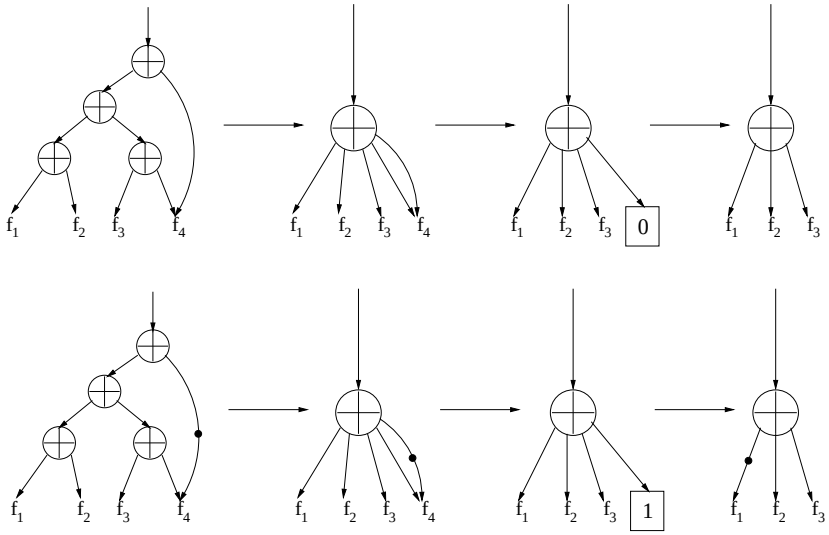


Figure 5.15: Additional Reductions for Meta-⊕-Nodes.

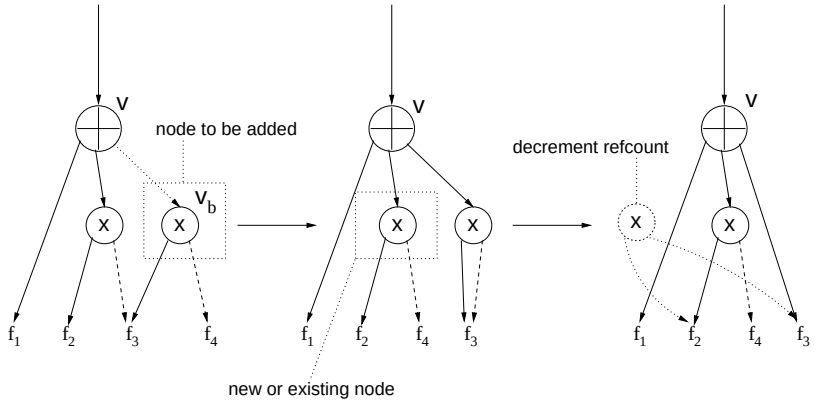


Figure 5.16: 2nd-Level-Reduction for Meta-⊕-Nodes.

Otherwise, if w.l.o.g. $f_n = \overline{f_{n+1}}$, then

$$f = \bigoplus_{i=1}^{n+1} f_i = \bigoplus_{i=1}^{n-1} f_i \ \oplus 1 = \overline{\bigoplus_{i=1}^{n-1} f_i}$$

(see Fig. 5.15). It is important to avoid the situation of inverting the incoming edges of f, because for this operation all nodes above the current variable level would have to be accessed and tested. This problem can be solved by inverting the first successor of f instead of inverting f itself, since if $f = f_1 \oplus f_2$, then $\overline{f} = \overline{f_1} \oplus f_2$.
Additionally, we have to take care of another important type of reduction that was already introduced for binary ⊕-nodes in the previous chapter(see Fig. 4.8 and Fig. 4.9). This reduction has to be tested, whenever a new successor node is put into the successor list of the meta-⊕-node v. Consider a branching node v_b that is supposed to be put into the successor list of v. Let v_b be labeled with the variable x and let v_{b_1} be its 1-successor and v_{b_0} be its 0-successor. n_{succ} denotes the actual number of successors of v. If now, any of the nodes v_i, $1 \leq i \leq n_{succ}$ that are already listed in the successor list of v, is labeled with the same variable x, then the following equivalences have to be tested:
Let v_{i_1} be the 1-successor of v_i and v_{i_0} the 0-successor of v_i, respectively. Then, if $v_{b_1} \equiv v_{i_0}$, v_i can be substituted by its successor v_{i_0} and instead of v_b another node v_b' labeled with variable x, with v_{i_1} as 1-successor and v_{b_0} as 0-successor has to be put into the successor list of v. In the best possible case, this node does already exist in the hash table and no new node has to be created. The reference count of the old nodes v_b and v_i is decremented and if no other node in the ⊕-OBDD is referencing them, they become obsolete and are not required anymore. The case for $v_{b_0} \equiv v_{i_1}$ has to be computed in the same way. See Fig. 5.16 for an illustration of this 2nd-level reduction for meta-⊕-nodes.
In general, if we want to employ meta-⊕-nodes, this necessitates a fundamental change in the ⊕-OBDD data structure. Up to now, the number of successors of a node was fixed. But from now on, when dealing with a variable number of successor nodes, the data structure has to be adapted accordingly and the

manipulation algorithms have to be adapted. In principle, there are two possible different choices of when to introduce them:

(1) **After synthesis**: Sum up all binary branching nodes after synthesis is completed.

(2) **During synthesis**: Perform synthesis with meta-$\oplus$-nodes directly right from the start.

Solution (1) can be easily implemented. Consider an already constructed $\oplus$-OBDD P. Starting from the root, we are traversing P levelwise. Note that this is possible since the hash table that is containing all the nodes is organized by levels. Each $\oplus$-node represents the possible root of a tree of $\oplus$-nodes. This tree is traversed in *depth-first-search* manner and all successing branching nodes will be put in the ordered successor list of the transformed root node, while the reference counter of all binary $\oplus$-nodes of the tree is accordingly decremented. Finally, the number of successors is determined and the new meta-$\oplus$-node is put into the appropriate unique table. Note that it is possible that the root node that has been transformed into a meta-$\oplus$-node has to be put into a different hash table, because reductions might have changed the reference to the next successing branching node variable. See Fig. 5.17 for the algorithm in pseudo code.
In order to realize approach (2) the entire synthesis procedures including the cofactor creation has to be reimplemented. A potential problem that might occur during the adjusted synthesis, if we are using meta-$\oplus$-nodes right from the start is that meta-$\oplus$-nodes possibly have to switch between different hash tables. This situation can occur each time when a new successor v_{new} is put into the successor array of the meta-$\oplus$-node v. If this new successor v_{new} is labeled with a variable index that is smaller than all other variable indexes of the nodes in the successor list $l(v_{new}) < \min_{1 \leq i \leq n} (v_{succ}[i])$, then, the $\oplus$-node v has to be relabeled and it must be put into the according hash table. But, if a node v with the same functionality f_v does already exist in the other hash table, we have two different nodes v and v', both representing the same Boolean function $f_v = f_{v'}$. To get rid of one of them, we would have to redirect all incoming edges of one of these nodes, what might cause that synthesis becomes a non local procedure, because all predecessors of the node that will be substituted have to be accessed.
Fortunately, for symbolic simulation, the $\oplus$-OBDD for a given circuit netlist description is constructed starting from the primary inputs by computing the corresponding $\oplus$-OBDD for every gate until all primary outputs are reached. In the synthesis procedure - independent from the chosen decomposition rule to be applied - a meta-$\oplus$-node that is to be created does only depend on already computed nodes. After its construction the node will never be a subject of change, because all nodes that still have to be created will not be in the transitional fanin of that particular node. Thus, if all predecessors are completely determined, a new meta-$\oplus$ can be created and be put into its corresponding hash table, without being changed anymore. This means that $\oplus$-OBDD synthe-

```
Input: ⊕-OBDD P_f with binary ⊕-nodes.
Output: ⊕-OBDD P'_f with meta-⊕-nodes.

assemble_meta_xor(P_f) {
   step through all nodes v of P_f levelwise {
     if (l(v) == ⊕) {
       v_new = transform_to_meta_⊕-node (v);
       v_new.n = create_successor_list(v_new.succ_list, v_new);
     }
   }
   return(P_f);
}

create_successor_list(v_new.succ_list, v) {
   // step in dfs order through tree rooted by v
   if (l(v) == ⊕) {
     n = create_successor_list(v_new.succ_list, v.1-succ) +
        create_successor_list(v_new.succ_list, v.0-succ);
   } else {
     put v into v_new.succ_list;
     n++;
   }
   return(n);
}
```

Figure 5.17: Algorithm for Transformation of Binary ⊕-Nodes to Meta-⊕-Nodes.

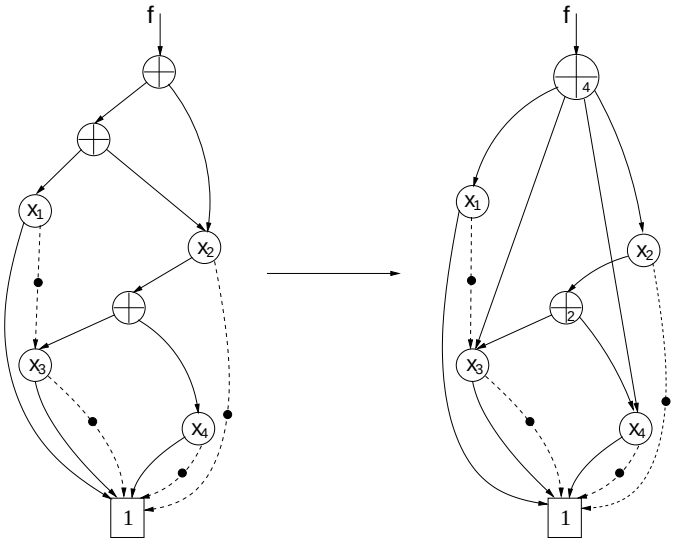


Figure 5.18: An Example for Meta-⊕-Node Transformation in a ⊕-OBDD.

sis based on meta-⊕-nodes is feasible, because all required operations remain local.
See Fig. 5.18 for an example of transforming an arbitrary ⊕-OBDD with binary ⊕-nodes into a ⊕-OBDD with meta-⊕-nodes. The number inside the meta-⊕-nodes denotes the number of successors.

⊕-OBDD Synthesis with Meta-⊕-Nodes

When the ⊕-OBDD synthesis procedure is adapted for the use of meta-⊕-nodes, independently of the chosen function expansion the following two important tasks have to be considered:

(1) the creation of cofactors for ⊕-OBDDs with meta-⊕-nodes as root nodes, and

(2) the creation of new meta-⊕-nodes in the synthesis procedure.

The creation of cofactors is the essential operation of the synthesis algorithm, because all the algorithms that are considered in this thesis are based on recursive procedure calls with the input function's cofactors as required parameter. For regular binary synthesis, the case of a tree of ⊕-nodes constituting the top of the ⊕-OBDD under consideration has to be taken special care of, because the situation requires the recursive application of the given binary cofactor algorithm and - as shown in the previous chapter - sometimes it also requires the duplication of several nodes of the ⊕-node tree.
The cofactor creation algorithm can be simplified for the usage of meta-⊕-nodes. If the cofactor P_{f_x} has to be created for a ⊕-OBDD P_f representing the Boolean function f w.r.t. the top variable x, and the root node of P_f happens to be a branching node v_b, the algorithm is not changed at all and one of the successor nodes of v_b will be returned.

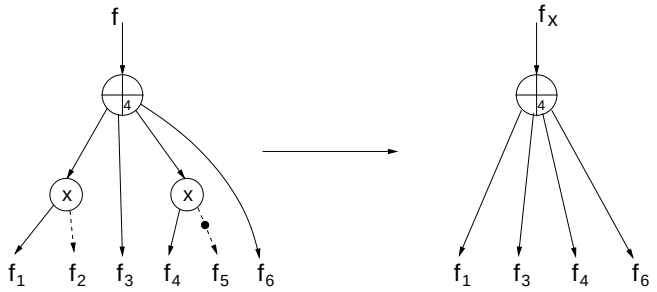


Figure 5.19: Cofactor Creation for a Meta-⊕-Node.

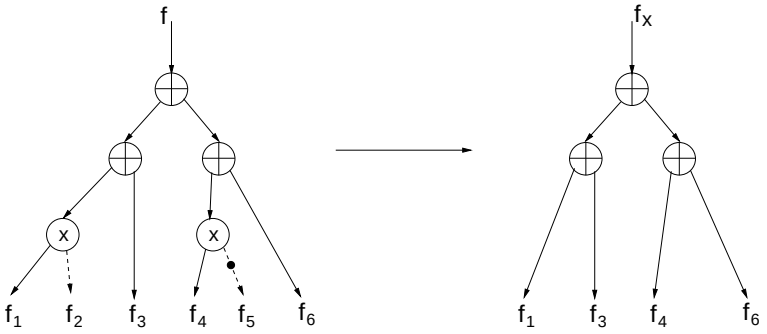


Figure 5.20: Cofactor Creation for a Tree of Binary ⊕-Nodes.

For the case that the top node of P_f is a meta-⊕-node $v_\oplus$, the algorithm works in the following way: By utilizing the concept of meta-⊕-nodes, we requested to follow the rule of never deploying more than one single meta-⊕-node on a path between any two branching nodes that are labeled with adjacent variables. Thus, all successor nodes that have to be considered for the algorithm are directly located in the successor list of the meta-⊕-node under consideration. The algorithm steps through the successor list and each node that is labeled with x will be replaced with its successor according to the chosen restriction $x = 1(0)$ (see Fig.5.19. For a comparison see Fig.5.20 showing the same situation with binary ⊕-nodes). Of course P_f has to be maintained and thus, for P_{f_x} one new meta-⊕-node has to be created, unless it does not already exist in the ⊕-OBDD. See Fig. 5.21 for an outline of the simplified cofactor creation algorithm for ⊕-OBDDs with meta-⊕-nodes.
Furthermore, if one of the successor nodes that are to be replaced happen to be a meta-⊕-node, the $v_\oplus$ has to be merged with that meta-⊕-node and the successor list of $v_\oplus$ has to be adapted accordingly for maintaining the required rule that between any two branching nodes of adjacent variables, there must be at most one single meta-⊕-node.
The recursive algorithms for ⊕-OBDD synthesis also have to be adapted for meta-⊕-nodes. Since, all described algorithms depend on a recursive call of themselves with the cofactors of their input functions, the recursion can remain unchanged, because we have already adapted the cofactor algorithm to meta-⊕-nodes. Only, if the application of the algorithm necessitates the creation of

```
Input: ⊕-OBDD P_f with meta ⊕-nodes, variable x, assignment a ∈ {0,1}.
Output: ⊕-OBDD P_{f_{x=a}}.

meta_cofactor(P_f, x, a) {
   if (top node of P_f is branching node) {
    if (top variable of P_f is x) {
     if ( a == 1) {
      return(1-successor of P_f);
     } else {
      return(0-successor of P_f);
     }
    } else {
     return(P_f);
    }
   } else { // top node of P_f is meta-⊕-node
    for all successors P_{f_i} of P_f do {
     if (top variable of P_{f_i} is x) {
      if ( a == 1) {
       new[i] = 1-successor of P_{f_i};
      } else {
       new[i] = 0-successor of P_{f_i};
      }
      if (new[i] is meta-⊕-node) {
       merge_⊕-nodes (P_f, new[i]);
      }
     } else {
      new[i] = P_{f_i};
     }
    }
   if (node(META-XOR, new) exists) {
    return(node);
   } else {
    node=create(META-XOR, new);
    return(node);
   }
}
```

Figure 5.21: Cofactor Creation Algorithm for a ⊕-OBDDs with Meta-⊕-Nodes.

a new $\oplus$-node $v_\oplus$, then, not a binary $\oplus$-node, but a meta-$\oplus$-node has to be initialized. If, one of the designated successors of $v_\oplus$ is a meta-$\oplus$-node, then the two nodes have to be merged as described in the transformation algorithm of Fig. 5.17. In this way, the synthesis algorithms can be adapted to meta-$\oplus$-nodes in a very simple way, unless the required transformation algorithms and the cofactor creation algorithm do already exist.

Experimental Results for $\oplus$-OBDDs with Meta-$\oplus$-Nodes

Now, for giving the proof that the introduction of meta-$\oplus$-nodes gives any advantage for $\oplus$-OBDDs, besides its possibility of developing simpler algorithms for their optimization, we have to compare $\oplus$-OBDD sizes for both models, when using binary and meta-$\oplus$-nodes. But, the plain node count is not a suitable measure for comparing these two variants anymore, because meta-$\oplus$-nodes that have more than two successors cannot be counted as a single node only. Thus, we have to refine the node count measurement. Here, for a better comparison, although implementation dependent, we have chosen the real size of the computed data structure in Bytes for measurement. The relation between the node count and the real size of $\oplus$-OBDDs using meta-$\oplus$-nodes can be computed in the following way:
Given a $\oplus$-OBDD P, let V_b be the set of branching nodes and $V_\oplus$ the set of meta-$\oplus$-nodes in P. Let $size_v$ the size of the node data structure and let $size_e$ be the size of a pointer to a memory address in our implementation. $succ(v)$ denotes the number of successors of node v. Then $|P|$, the size of P can be computed by

$$|P| = (|V_b| + |V_\oplus|) \cdot size_v + \sum_{v \in V_\oplus} succ(v)) \cdot size_e$$

By comparing $|P|$ for $\oplus$-OBDDs of both variants, we can estimate the benefit of the additional reduction possibilities. In Table 5.6 a comparison of $\oplus$-OBDD sizes with binary $\oplus$-nodes versus meta-$\oplus$-nodes for an arbitrary selection of circuits of our benchmark set is given. See Tables A.12 and A.13 in the Appendix for the results of the complete benchmark set.
In the first column the name for the benchmark circuit is given that should be subject of symbolic simulation with $\oplus$-OBDDs, while in the second and the third column $|V|$ for the case of binary and meta-$\oplus$-nodes is listed. For this set of experiments the variable order given inherently with the circuit description was used and as decomposition method we have chosen exclusively pDE, because many $\oplus$-nodes should be created (for nDE the results are rather similar).
As we can see in the table, the size of the $\oplus$-OBDD when using meta-$\oplus$-nodes, ranges from 24% up to 146% of the original size, by an average reduction to 87%. This reduction in size can be explained because of the following two facts:

(1) With binary $\oplus$-nodes, for realizing an XOR-operation over $m \in \mathbb{N}$ addends, we have to create a tree of $m-1$ $\oplus$-nodes and thus, creating a possible overhead of administration information. The same XOR-operation can be realized by a single meta-$\oplus$-node with m successors and thus, we

circuit	⊕-OBDD size [Bytes]	
	binary ⊕-nodes	meta-⊕-nodes
s499	23040	24128 [104%]
s444	14040	11136 [79%]
s1488	47376	34332 [72%]
s1423	4824288	3935156 [81%]
s1269	1437192	1101680 [76%]
comp	30959568	27142104 [87%]
mm9a	19213452	14583716 [75%]
mux	7812	1960 [25%]
cm150a	7920	1960 [24%]
apex1	320436	197496 [61%]
alu32r	670644	460920 [68%]
C880	11861424	8576036 [72%]
C499	493164	670188 [135%]
Σ	**181.114.668**	**158.147.088 [87,3%]**

Table 5.6: Effects of the Meta-⊕-Nodes and Additional Reduction Rules on ⊕-OBDD Size.

only require m additional 32-bit addresses for each successor instead of creating m complete binary nodes.

(2) The second reason for the possibility of achieving smaller ⊕-OBDD sizes by using meta-⊕ nodes lies in the potential of performing additional reductions on meta-⊕ nodes with $m > 2$ successors, as we have already mentioned in the previous section.

But, 7 out of the 65 benchmarks for meta-⊕ nodes are producing a slightly larger result. This effect can also be explained easily. The just described memory saving does only occur for a tree of m binary ⊕-nodes, if non or only a small number of its intermediate nodes is referenced from other nodes of the ⊕-OBDD. For each referenced intermediate node, an additional meta-⊕ node has to be created and thus, additional memory has to be utilized. In some cases, as e.g. for $C499$ this results in a ⊕-OBDD that is 35% larger than the original one.
But, all in all, the use of meta-⊕ nodes results in an average reduction of ⊕-OBDD size to 87% of the original size.
For a comparison, we also did symbolic simulation of the benchmark set, where meta-⊕-nodes where enabled during synthesis already. As expected, the achieved results did not differ very much from the results achieved, when meta-⊕-nodes are introduced, after synthesis has finished. The difference is caused by possible reductions that can already take place during synthesis and cannot be found in the binary version. These additional reductions might cause that the represented function will be constructed out of subfunctions that are different from the representation of the same function with binary ⊕-nodes. The

achieved sizes are comparable or even sometimes slightly better compared to the first method. Also the computation time of both methods is almost equal. Although for method (1) the creation of meta-$\oplus$-nodes requires an additional walktrough, this extra time is also necessary for the more complex reduction rules in approach (2). Compared to the synthesis time for $\oplus$-OBDDs, the additional time required for the transformation of $\oplus$-nodes into meta-$\oplus$-nodes is neglectible. See Table A.14 A.15 and in the Appendix for an overview of the achieved results for synthesis with meta-$\oplus$-nodes.

Dynamic Placement of Meta-$\oplus$-Nodes

Now, that we have shown the effects of meta-$\oplus$-nodes on the size of $\oplus$-OBDDs, again the question of how many meta-$\oplus$-nodes should be employed and where should they be positioned within the $\oplus$-OBDD has to be answered. While on the one hand we can directly transfer our results from the benchmark testing with binary $\oplus$-nodes for determining an appropriate number of $\oplus$-nodes to be introduced, on the other hand for finding the right position of the meta-$\oplus$-nodes, we have to show that it is also possible to move meta-$\oplus$-nodes up and down by one position efficiently in the $\oplus$-OBDD.
First, let's start with moving a meta-$\oplus$-node downwards. Given a $\oplus$-OBDD P representing the Boolean function f_P, with the meta-$\oplus$-node v as the root node. P is ordered following a given variable order π and for the variables x_i, x_j, x_k it holds that $x_i < x_j < x_k$. W.l.o.g. let v be a meta-$\oplus$-node with $n = 3$ successors. The given swap routine holds for an arbitrary number $n \in \mathbb{N}$ of successors. Let the three successors v_1, v_2, and v_3 of v be branching nodes labeled with x_i, x_j, and x_k, respectively. Now, we exchange the meta-$\oplus$-node v with its successor x_i, which comes first w.r.t. the variable order π.
For an efficient implementation, v itself is only relabeled and transformed into a branching node that will be labeled with x_i and connected to its two new successors $v_{f_{x_i}}$ and $v_{f_{\overline{x_i}}}$, both being meta-$\oplus$-nodes with $n = 3$ successors each, at least if no reduction rule can be applied. $v_{f_{x_i}}$ is connected with the 1-successor of v_1, or in the arbitrary case with every 1-successor of a former successor of v that is labeled with x_i, and with v_2 and v_3, i.e. every other former successor of v that is not labeled with x_i. $v_{f_{\overline{x_i}}}$ then is connected with the 0-successors of all former successor nodes of v that are labeled with x_i and with all other former successors of v that are not labeled with x_i. For a better understanding of the *meta-$\oplus$-swap-down* procedure see Fig. 5.22 and a sketch of the algorithm in pseudo code is given in Fig. 5.23. For simplicity reduction procedures and rules for working with complemented edges have been left out. When working with complemented edges and one of the successors v_i of v that are labeled with the same variable x_i is inverted, then this inversion will be passed to both the successors of v_i according to the complement equivalence rules for branching nodes (see Fig. 2.6). If any other successor of v is inverted, this inversion will simply be kept (see Fig. 5.24) .
But what, if a successor node of the branching nodes v_i that are labeled with the variable to be exchanged is a meta-$\oplus$-node? By performing the regular exchange routine, we would end up with two meta-$\oplus$-nodes on a path between

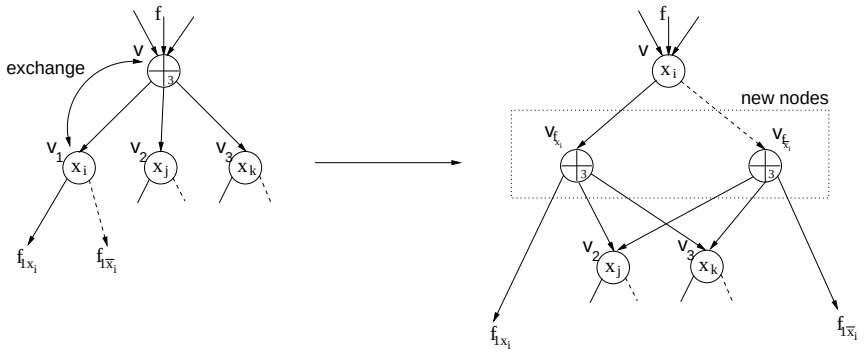


Figure 5.22: Meta-⊕-Node Swap Down Operation.

Input: ⊕-OBDD P_f with top node v
Output:⊕-OBDD P_f with top node and successing nodes being exchanged

```
swap_down(P_f) {
  if (v is meta-⊕-node) {
    nsucc = v → n;
    label = min(l(v_{f_1}), ..., l(v_{f_nsucc}));
    create meta-⊕-nodes v_1, v_2 with nsucc successors;
    for all successors i do {
      if (l(v_{f_i}) == label) {
        f_1 = v_{f_i} → then; f_2 = v_{f_i} → else;
      } else {
        f_1 = f_2 = v_{f_i};
      }
      if (f_1 is meta-⊕-node) {
        merge meta-⊕-nodes v_1 and f_1;
      } else {
        put f_1 in successor list of meta-⊕-node v_1;
      }
      if (f_2 is meta-⊕-node) {
        merge meta-⊕-nodes v_2 and f_2;
      } else {
        put f_2 in successor list of meta-⊕-node v_2;
      }
    }
    decrement reference count of v_{f_1}, ..., v_{f_nsucc};
  }
  return(v);
}
```

Figure 5.23: Sketch of the Algorithm for Downward Exchange of Meta-⊕-Nodes and Branching Nodes.

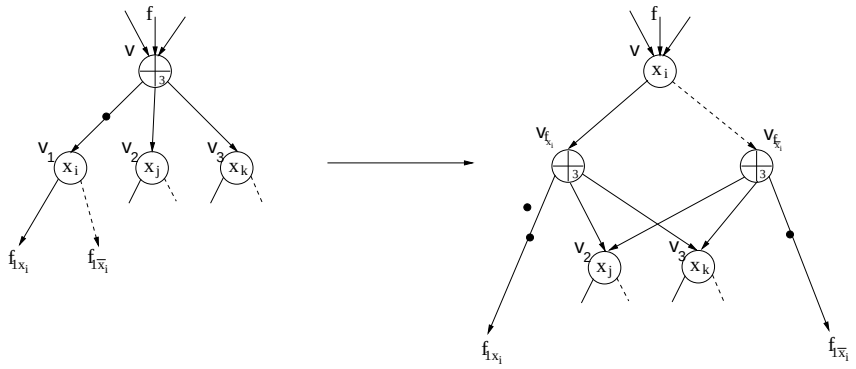


Figure 5.24: Meta-⊕-Node Swap Down Operation with Complemented Edges.

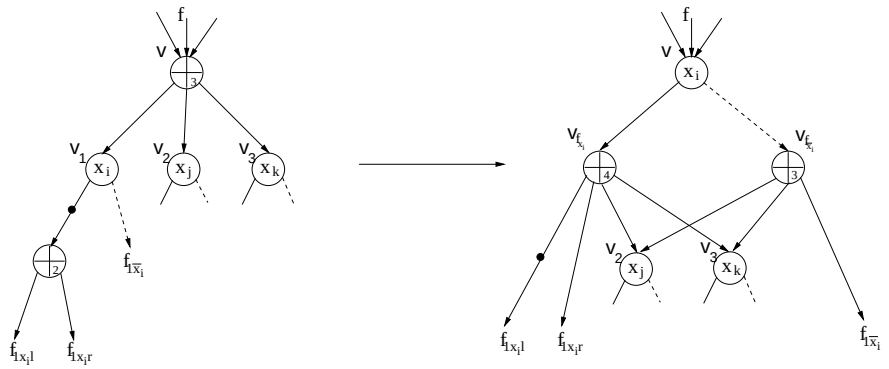


Figure 5.25: Extended Meta-⊕-Node Swap Down Operation with Complemented Edges.

two variables. This does not fit the given requirement of not allowing more than a single meta-⊕-node between two variables on any given path in the ⊕-OBDD. Thus, the exchange procedure has to be extended by joining meta-⊕-nodes automatically that occur as being adjacent after an exchange operation. See Fig. 5.25 for the extended *meta-⊕-node-swap-down* operation.
If we are working with complemented edges, for those complemented edges pointing to meta-⊕-nodes the complement is passed to the first successor of the meta-⊕-node under consideration due to the complement equivalence rules for ⊕-nodes (see also Fig. 5.25).
For exchanging a meta-⊕-node with an adjacent branching node *upwards*, other special cases have to be considered. First, assume that we want to exchange a meta-⊕-node $v_\oplus$ with its predecessor branching node v that is labeled with variable x_i.
W.l.o.g. let us assume that $v_\oplus$ is the 1-successor of v. If $v_\oplus$ has $i \in \mathbb{N}$ successors, then v is transformed into a meta-⊕-node and additional memory for i possible successors is allocated. Then, for each node v_k, $1 \leq k \leq i$ of these i potential successor nodes, the 1-edges are connected to the k-th successor of $v_\oplus$, which will be denoted as $v_{\oplus_i}$. If the 0-successor of v is also a meta-⊕-node $v'_\oplus$, then the number i of possible successor nodes of v computes to the maximum of

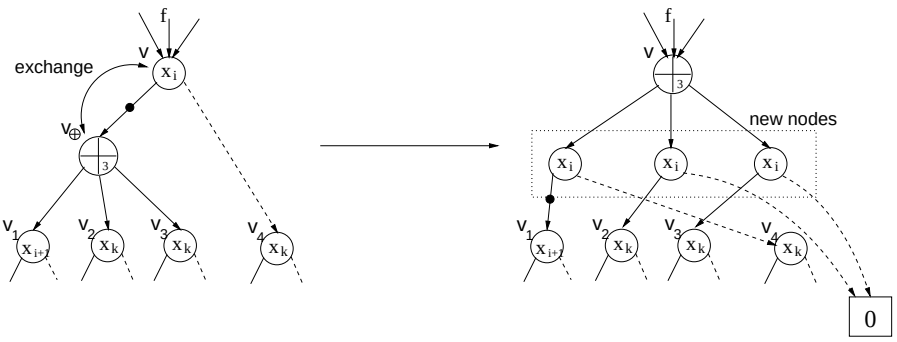


Figure 5.26: Meta-⊕-Node Swap Up Operation.

the successors of $v_\oplus$ and $v'_\oplus$, $i = \max(v_\oplus.n,\ v'_\oplus.n)$. Only in that case the 0-successors of the i new nodes v_i are connected to the successors of $v'_\oplus$.
If $v_\oplus.n > v'_\oplus.n$, then we will run out of successors for $v'_\oplus$. For those new branching nodes v_k that are affected by this case and that don't have received a 0-successor from $v'_\oplus$, the 0-edge will be connected to the 0-sink. For $v'_\oplus.n > v_\oplus.n$ the same procedure has to be carried out for the 1-successors of v_k, respectively. See Fig. 5.26 for an illustrating example of the *meta-⊕-node swap-up* operation. A simplified sketch of the algorithm in pseudo code is given in Fig. 5.27 and Fig. 5.28.
If complemented edges are used, the same equivalences as for the swap down procedure are applied (see also Fig. 5.26 for an example).
Note that the routine that is taking a new successor v_k into the successor array of a meta-⊕-node v takes care of possible reductions and adjusts dynamically the value $v.n$, the number of successors of v.
Another exception for the case of swapping a meta-⊕-node upwards occurs, if the initial branching node v that should be exchanged with its successor meta-⊕-nodes, itself is the successor of some meta-⊕-node v_{sup}. Due to the fact that v keeps the same variable label also after the node exchange, also its predecessor v_{sup} does not have to change its hash table position. But, after the exchange, v has become a meta-⊕-node, while being the successor of another meta-⊕-node. To maintain the property that between any two adjacent branching nodes on a path in the ⊕-OBDD, there must be at most one meta-⊕-node, we have to join these two meta-⊕-nodes. This is done by simply merging the successor array of v into the successor array of v_{sup} and by decrementing v's reference count. But, v may also be referenced by other branching nodes and thus, the original node v has to be kept in the ⊕-OBDD (see Fig.5.29 for an illustration). Note that after merging v_{sup} and v the operation is not reversible anymore.
To perform the swap operation for ⊕-OBDDs with deploying meta-⊕-nodes, it is necessary to scan all predecessors of v to find out, whether there is a meta-⊕-node among them. For reasons of efficiency and to avoid that this operation has to be repeated for every single swap up operation for a complete variable level, the merge operation is only performed once after all meta-⊕-nodes of that level have been swapped.
The merge operation traverses the ⊕-OBDD in a *dfs* manner, but, only down

```
Input: ⊕-OBDD P_f with top node v
Output:⊕-OBDD P_f with top node and successing nodes being exchanged

swap_up(P_f) {
  if (v is branching-node labeled with x_i) AND
     ((v.1-succ is meta-⊕-node) OR
      (v.0-succ is meta-⊕-node) {
    n_then = # succ of v.1-succ;
    n_else = # succ of v.0-succ;
    n = max(n_then , n_else);
    transform v to meta-⊕-node with n successors;
    for all n new successors v_i of v do {
     if (v.1-succ is meta-⊕-node) {
      if ( i ≤ n_then ) {
       then = (v.1-succ).succ_i;
       if (v.1-succ is complemented AND (i == 1) ) {
        then = complement_node(then);
       }
      } else {
       then = 0-sink;
      }
     } else {
      then = (v.1-succ);
     }
```

Figure 5.27: Sketch of the Algorithm for Upward Exchange of Meta-⊕-Nodes and Branching Nodes (Part 1).

```
Input: ⊕-OBDD P_f with top node v
Output:⊕-OBDD P_f with top node and successing nodes being exchanged

      if (v.0-succ is meta-⊕-node) {
        if ( i ≤ n_else ) {
          else = (v.0-succ).succ_i;
          if (v.0-succ is complemented AND (i == 1) ) {
            else = complement_node(else);
          }
        } else {
          else = 0-sink;
        }
      } else {
        else = (v.0-succ);
      }
      if ( then == else ) {
        v_i = then;
      } else {
        v_i = create_new_node(then, else, x_i);
      }
      put v_i in successor list of v;
    }
    decrement reference count of v.1-succ;
    decrement reference count of v.0-succ;
  }
  return(v);
}
```

Figure 5.28: Sketch of the Algorithm for Upward Exchange of Meta-⊕-Nodes and Branching Nodes (Part2).

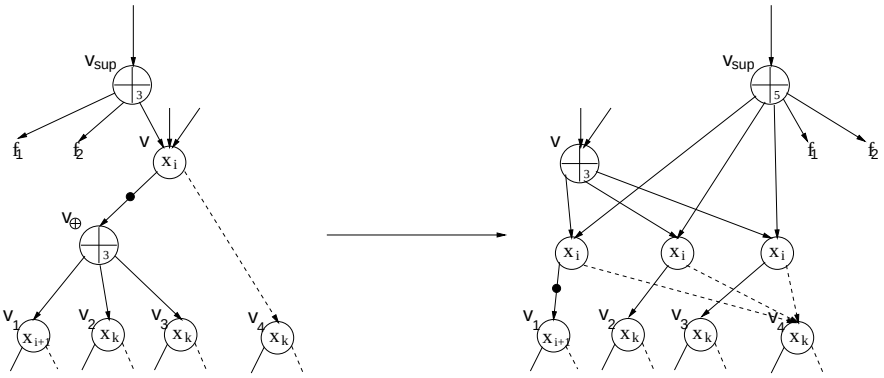


Figure 5.29: Joining Meta-⊕-Nodes After Swap Up Operation.

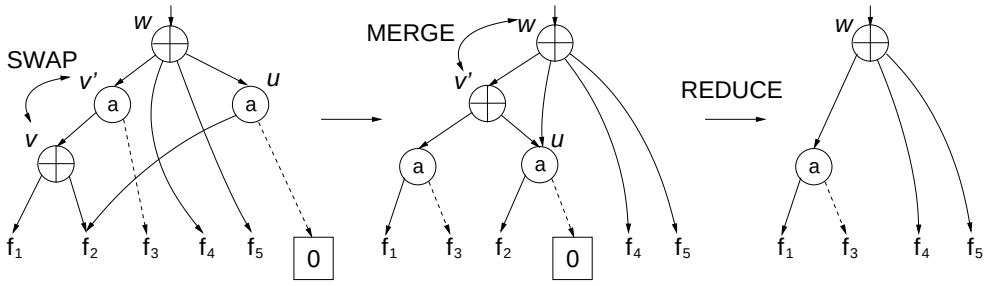


Figure 5.30: Non Reversible SWAP Operation.

to variable level $i-1$. Then, all meta-$\oplus$-nodes v_{sup} are considered that include a pointer to a meta-$\oplus$-node v in their successor array, which must be of level i. v_{sup} and v are then subject to the merging procedure. For avoiding additional time overhead during construction, successor nodes of $\oplus$-nodes can be marked by setting a spare bit in the data structure of the node. If this particular bit is not set for a node being involved in a swap-up operation, none of its predecessors have to be scanned.

5.2.5 Jiggling - A Simple Heuristic for $\oplus$-Node Placement

Now that we are able to perform the exchange of the position of already introduced meta-$\oplus$-nodes in the $\oplus$-OBDD data structure in an efficient way, this $\oplus$-node swap algorithm can be further utilized for $\oplus$-OBDD minimization. A heuristic for $\oplus$-OBDD minimization should make use of local effects achieved by considering the swap of a single $\oplus$-node. If this swap operation results in a positive effect, i.e. does it decrease the $\oplus$-OBDD size, then this swap operation should be confirmed or reversed, otherwise. But, is it really necessary to perform that expensive swap operation in both directions, if it is possible that no benefits will be achieved by its application?
On the other hand, if we keep the fixed rule that between two adjacent branching nodes, there must be at most one meta-$\oplus$-node, the situation that two meta-$\oplus$-nodes happen to be adjacent after performing a swap operation can only be resolved by merging them to a single meta-$\oplus$-node. Note that this merge operation is also necessary for achieving all reductions possibilities. But, if, after a successful merge operation no reduction in $\oplus$-OBDD size can be achieved, the split of two already merged $\oplus$-nodes will be difficult, because additional information about the former state of the two meta-$\oplus$-nodes has to be memorized. Thus, in principle, this operation is not reversible, if we don't store some copy of the original meta-$\oplus$-nodes. See Fig. 5.30 for an illustration: After a swap-up operation for meta-$\oplus$-node v (1) the resulting meta-$\oplus$-node v' (2) has to be merged with the predecessor meta-$\oplus$-node w above (3). For considering any changes in size, reductions have to be carried out and thus, information on the sub-$\oplus$-OBDD rooted by node u is lost.
The simple solution of that problem could be a simulation of the swap operation, without really performing it's effects on the involved nodes. By only simulat-

ing this operation, we are able to compute the $\oplus$-OBDD size after the swap operation and only if the achieved result is smaller compared to the previous size, the swap operation finally can be performed, because now, the expenses in terms of additional computation time are worth while.
By performing only swap operations that are really leading to some reductions in size, we have designed another simple greedy heuristic for achieving some local minimum in $\oplus$-OBDD size by changing the position of already created $\oplus$-nodes. The efficiency of the algorithm can be adapted by using a threshold parameter α that decides, whether a swap operation will be carried out or not. Let $|P_{new}|$ denote the size of the $\oplus$-OBDD after a possible swap operation and let $|P_{old}|$ denote the original size, respectively. Then, the proposition that has to be fulfilled for performing a swap operation is

$$\text{do swap, if } |P_{new}| \leq \alpha \cdot |P_{old}|.$$

If $\alpha = 1$, then the original designed greedy algorithm is carried out. If $\alpha < 1$, then a swap will only be performed, if the achieved reduction in size is more relevant. Otherwise, if $\alpha > 1$, the algorithm is no longer greedy anymore and also swaps that don't directly lead to a reduction in size will be carried out. This gives way for designing an algorithm that is also able to leave an already achieved local minimum again, but, on the other hand, the $\oplus$-OBDD might also grow in size again.
The algorithm presented in this section is called *jiggle*-algorithm. This is due to the fact, that for each meta-$\oplus$-node of the $\oplus$-OBDD we try to find a better position by jiggling the node up and down by one position. The jiggle-algorithm receives the threshold factor α as an input parameter and steps through each level i of the $\oplus$-OBDD starting at the root level. For each meta-$\oplus$-node v in level i a swap_up operation is simulated and if the computed resultant size $|P_{new}|$ is smaller than the original size $|P_{old}|$ multiplied by the threshold factor α, the swap_up operation really will be carried out for v. Otherwise, a swap_down operation is simulated for v and also will only be carried out, if the resultant size $|P_{new}|$ is smaller than the original size $|P_{old}|$ multiplied by the threshold factor α. See Fig.5.31 for an outline of the jiggle-algorithm in pseudo code.
For our experimental setup, first, we constructed the $\oplus$-OBDD from the given circuit description and in the next step we applied the jiggle algorithm for the entire graph until no further improvement could be achieved. Usually after 3 or 4 rounds, no further improvement could be achieved anymore.
In Table 5.7 selected results of the jiggle heuristic applied to our benchmark set are listed. The first column denotes the circuit's name, in the second column the circuit's OBDD size is given for a reference. The starting point for the experiments was the circuit's $\oplus$-OBDD representation based on exclusive application of pDE with meta-$\oplus$-nodes. The size of this $\oplus$-OBDD is given in the third column. Column 4,5, and 6 list the achieved $\oplus$-OBDD sizes for the application of the jiggle heuristic for one, two, and three rounds. The upper part of the table contains results for sequential circuits, while the lower part comprises the results for combinatorial circuits. The last row gives the overall sums of the achieved representation sizes. The percentage of the overall sums for each

```
Input: ⊕-OBDD P_f with top node v and threshold factor α
Output:⊕-OBDD P'_f

jiggle(P_f, α ) {
  for all levels i do {
    for all nodes v in level i do {
      if (v is meta-⊕-node) {
        new = size of P_f after simulated swap_up(v);
        if ( new < α·actual size of P_f) {
          swap_up(v);
        } else {
          new = size of P_f after simulated swap_down(v);
          if ( new < α·actual size of P_f) {
            swap_down(v);
          }
        }
      }
    }
  }
  return(P_f);
}
```

Figure 5.31: Sketch of the Jiggle-Algorithm for ⊕-OBDD Minimization.

column are given w.r.t. the achieved exclusive pDE ⊕-OBDD representation size given in column three. See Tables A.18 and A.19 in the Appendix for the complete results for all combinatorial and sequential circuits of the benchmark set.

A dash inside a cell of the table denotes that for this round the jiggle heuristics did not achieve any further improvement. Note that due to the given resource limitations for the following six circuits the jiggle heuristic could not finish successfully: *C880, comp, rot, mm9a, mm9b, mult16a.*

In the average the first application of the jiggle heuristics results in a 9.3% gain in the representation size. The next round of the jiggle algorithm wins additional 5.1%, while the last application is only able to contribute an additional gain of less than 0.1%. Thus, the overall improvement in size comes to 14.4% in the average. While for single circuits as *s*499 the improvement of the first application of jiggle is up to about 50%, for most of the other circuits the improvement is much less.

Due to the strictly greedy nature of the algorithm the search space for positioning the ⊕-nodes is rather limited. Also the exclusive pDE ⊕-OBDD representation as a starting point is not necessarily the best choice for achieving the smallest possible result that can be achieved by jiggle. If we start with a ⊕-OBDD representation based on the local greedy algorithm from the previous section, the final ⊕-OBDD sizes are much smaller, but the improvement that

Circuit		⊕-OBDD size [Bytes]			
	OBDD	pDE-meta	Jiggle (1)	Jiggle (2)	Jiggle (3)
sbc	133740	135892	117584	117220	-
s635	23616	64692	42688	-	-
s499	12096	24128	12964	-	-
s420.1	9440172	19932	16436	16412	-
s1494	36576	36196	31068	29688	-
s1423	3544344	3938476	3360020	3341752	-
8085	4242276	2614032	2255804	2253144	2253004
x3	99360	76232	64520	-	-
my_adder	11796372	18873336	16252724	13369124	-
mux	4718556	2240	1896	-	-
i8	157176	404896	283212	202136	-
i4	15156	37320	22920	15164	-
frg2	232956	142604	119036	-	-
count	8424	10068	8836	-	-
cm150a	4718556	2280	1876	-	-
bw6x6	29880	96592	87752	86652	-
apex1	1020096	197656	187108	185164	-
alu4	42552	53772	40064	38108	34448
alu32r	6813576	460920	334264	-	-
adsb32r	19008	47448	41624	38072	-
adder16	11801232	21629848	21238784	-	-
C432	62388	184080	83044	-	-
C1908	1296252	1163384	1129004	1109496	1108708
Σ	**72.843.732**	**59.273.572**	**53.788.556**	**50.719.284**	**50.712.344**
	122.9%	100%	90.7 %	85.6%	85.6%

Table 5.7: Jiggle Heuristic for Dynamic ⊕-Node Placement

circuit	Overal Runtime		
	OBDD	⊕-OBDD	
		pDE-meta	Jiggle & pDE-meta
Σ	66.9	1813	2509
normalized	**1**	27.1	37.5

Table 5.8: Time Requirements of the Jiggle Heuristic

is gained in addition by the jiggle algorithm is much less. This is due to the fact that the local greedy algorithm already computes well suited positions for the employed ⊕-nodes and jiggle is not able to obtain a significant additional reduction.

If we take a look at the time requirements for the jiggle heuristic, it is clear that if it is applied after the synthesis process is over, the time for the heuristic adds to the time required for synthesis. If we compare the time required for ⊕-OBDD synthesis with and without applying the jiggle heuristics afterwards, the required time overhead is 38.5% in the average (see Table 5.8). Note that with the additional jiggle algorithm, the computation of 7 circuits of the original benchmark could not be finished due to resource limitations. Also note that synthesis, if performed while using meta-⊕-nodes is slightly faster compared to synthesis with binary ⊕-nodes.

Certainly, proceeding in this way is not rather suitable. But, the experiment could show that already introduced $\oplus$-nodes can be repositioned for achieving a smaller representation size, although the achieved improvement in size is in the average only about 14.4%. Although this improvement obtained by the jiggle algorithm is not really significant, it might be rather useful if it can be applied during synthesis in the same way as dynamic variable reordering. In this way, peak memory sizes can be avoided that are exceeding given resource limitations, otherwise. For applying the heuristic dynamically during the synthesis procedure, certain prerequisites have to be followed that will be summarized in the upcoming chapter.
Another possibility for achieving smaller $\oplus$-OBDD sizes, while on the other hand showing a more significant effect of the jiggle heuristic might be the random insertion of a certain fraction of $\oplus$-nodes, as already performed in a previous section of this work. Then, the positions of these randomly inserted $\oplus$-nodes can be improved by the jiggle heuristic afterwards.

Dynamic Jiggle Algorithm

One way of further using the concept of the jiggle algorithm is employing the algorithm dynamically during $\oplus$-OBDD synthesis. The purpose of applying the jiggle algorithm during synthesis is to enable the construction of $\oplus$-OBDDs that can not be computed in the regular way within the given resource limitations. By reducing the size of the already constructed part of the $\oplus$-OBDD under consideration, we are able to continue synthesis beyond the previously stated resource limits.
In principle the concept of the *dynamic jiggle algorithm* can be defined in the following way:
Let P be the $\oplus$-OBDD that is to be constructed and let $\alpha \in \mathbb{N}$, $\alpha > 0$ be a fixed threshold value.

(1) Start regular $\oplus$-OBDD Synthesis of $|P|$.

(2) If $|P| > \alpha$ interrupt the synthesis procedure and start the jiggle algorithm for optimizing $|P|$.

(3) Continue the synthesis procedure for $|P|$, chose a new threshold value $a' > a$, and continue with step (2).

To enable the jiggle algorithm for being utilized dynamically use some prerequisites have to be fulfilled:

- **Synthesis**:
  The $\oplus$-OBDD synthesis algorithm has to employ meta-$\oplus$-nodes right from the start. If we are working with binary $\oplus$-nodes in synthesis, the transformation algorithm for meta-$\oplus$-nodes would have to be called, before the jiggle algorithm can take place and afterwards a retransformation from meta-$\oplus$-nodes into binary $\oplus$-nodes would be necessary. To avoid this additional overhead, the synthesis algorithm is utilizing meta-$\oplus$-nodes

right from the start. After the jiggle algorithm has been successfully applied, the caches for storing already computed synthesis results have to be cleared, because during the optimization of the $\oplus$-OBDD memory addresses of single nodes can be changed and the result stored in the cache is no longer valid.

- **Jiggle Algorithm**:
  Before the jiggle algorithm can be performed, a garbage collection has to be conducted to get rid of dead nodes that are not used any longer in the current $\oplus$-OBDD. It would be of no sense to include these nodes into the jiggle algorithm, because a change of their position does not affect the actual size of the $\oplus$-OBDD under consideration. Before starting the synthesis again, we have to take care that all rules concerning the usage of meta-$\oplus$-nodes in $\oplus$-OBDDs are really fulfilled. I.e., possible merges of meta-$\oplus$-nodes that happen to be adjacent after the application of the jiggle algorithm have to be carried out and all meta-$\oplus$-nodes have to be in the correct variable table according to the reference to their first successor branching node.

Following these prerequisites it is possible to apply the jiggle algorithm during the synthesis procedure. For the successful application of the dynamic jiggle algorithm the chosen value for the threshold value α is also important as well as the question of how to change α after the threshold value has been exceeded. Here, we follow the usual strategy that is deployed during dynamic variable reordering for OBDDs [Som96]. In dynamic variable reordering for OBDDs, if the threshold value is exceeded, for the next round of synthesis, the threshold value simply is multiplied by a factor $k = 2$. Thus, for each round i, $i > 0$ of the $\oplus$-OBDD synthesis, the threshold value α_i changes in the following way:

$$\alpha_i = \alpha_{i-1} \cdot k,$$

while $\alpha_0 \in \mathbb{N}$ is chosen arbitrarily before the synthesis starts. Other variants as e.g., the addition of a constant factor or the multiplication with a value $k > 2$ are not as efficient the multiplication by $k = 2$. More sophisticated strategies concerning the behavior of the threshold factor and the effect on $\oplus$-OBDD size and computation time are subject to further research.
See Table 5.9 for selected results from the benchmark set for the application of the dynamic jiggle algorithm (For an overview of the complete results see Tables A.18 and A.19 in the Appendix). The first column denotes the circuit names, while in column two, as a reference, the OBDD size for the circuit is listed. Column three lists the size of the $\oplus$-OBDD for the given circuit achieved with the dynamic jiggle heuristic applied during $\oplus$-OBDD construction. For all the experiments the threshold seed value was fixed to $\alpha_0 = 100000$ and the threshold factor was set to $k = 2$. For further optimization the regular jiggle algorithm is applied and the results are given in the columns four to six. In the last row the overall sums for all combinatorial and sequential circuits are given and the percentage in relation to the original OBDD size. Note that OBDD- and $\oplus$-OBDD-sizes are both given in Bytes. A dash in a cell of the

Circuit		⊕-OBDD size [Bytes]			
	OBDD	pDE-DynJiggle	Jiggle (1)	Jiggle (2)	Jiggle (3)
sbc	133740	135452	120152	119400	-
s635	23616	69032	46092	-	-
s499	12096	24128	12964	-	-
s420.1	9440172	19932	17812	-	-
s1494	36576	36068	32028	31856	-
s1423	3544344	3937408	3622180	3561860	-
8085	4242276	4376740	4242916	4187524	-
mm9a	26487648	21406700	21405004	-	-
mm9b	30540926	24485432	24482980	-	-
mult16a	12975912	23822712	19286296	-	-
x3	99360	77636	71184	71064	-
my_adder	11796372	18854096	14486324	14486056	-
mux	4718556	1960	1896	-	-
i8	157176	404736	283040	283804	-
i4	15156	37320	23864	23704	-
frg2	232956	139140	120532	120292	-
count	8424	10068	8836	-	-
cm150a	4718556	1960	1876	1876	-
bw6x6	29880	95752	90616	90252	-
apex1	1020096	201152	182324	-	-
alu4	42552	53740	38904	38712	-
alu32r	6813576	460920	364968	-	-
alu32	438984	31628	28464	-	-
adsb32r	19008	47448	39692	-	-
adder16	11801232	21467152	20836304	20823676	-
C432	62388	186136	175592	-	-
C880	12456	10571900	10291912	10170540	10085944
C1908	1296252	1162736	1240312	-	-
comp	16513128	19746056	19663748	18741360	18741216
rot	36166674	7705076	7112688	6918432	-
Σ	**194.146.880**	**168.593.280**	**156.974.776**	**155.555.840**	**155.461.524**
	100.0 %	86.6 %	80.9 %	80.1 %	80.1 %

Table 5.9: Dynamic Application of the Jiggle Heuristic.

circuit	Overal Runtime				
	OBDD	⊕-OBDD			
		pDE-meta	pDE-meta & Final Jiggle	pDE meta & Dyn Jiggle	pDE meta & Dyn Jiggle & Final Jiggle
Σ	91.87	2563	-	2508	3463
norm	**1**	27.9	(37.5)	27.3	37.7

Table 5.10: Time Requirements for Dynamic Jiggle Heuristic.

table denotes that no further improvement by applying the jiggle heuristic was possible anymore.
Now, the six circuits *C880, comp, rot, mm9a, mm9b, mult16a* that could not be fully computed with the regular jiggle algorithm can be computed with the dynamic jiggle approach within the given resource limitations. This confirms the obvious assumption that applying the optimization routine during construction of the $\oplus$-OBDD decreases the required peak memory sizes.
On the other hand concerning the final achieved size the results of the application of the dynamic jiggle algorithm are almost comparable with the regular jiggle algorithm. Because the optimization already did take place during construction of the $\oplus$-OBDD the achieved sizes are related to the OBDD size and here the overall reduction in sizes ranges from 13.4 % up to 19.1 % in the average for dynamic jiggle with further optimization by regular jiggle.
If we compare the time requirement for synthesis with and without the application of the dynamic jiggle heuristic (see Table 5.10), we observe that almost no additional time is required, compared to the standard synthesis. Note that only one circuit of the original benchmark set could not be computed due to resource limitations. The avoidance of memory peaks and the related loss in management overhead is responsible for that result. If less nodes have to be managed by the $\oplus$-OBDD package, runtime also decreases and the additional time required for the jiggle heuristic does does not affect the runtime behavior compared to the regular synthesis procedure. Of course, if we apply a Final Jiggle step, after the synthesis procedure has ended, the overall required time is comparable to the experiments in the previous section, where we did regular synthesis and final jiggling. Again, the runtime is still not comparable to the runtime achieved for OBDDs with the CUDD package.
The major advantage of the dynamic application of this techniques comes due to the fact that it is now possible to apply the algorithm to circuits that could not be successfully computed before, because huge peak memory sizes can be avoided during the synthesis procedure.

5.2.6 Applications of Dynamic $\oplus$-Node Placement

By using the procedures for moving $\oplus$-nodes we are able to design a simple algorithm to transform any $\oplus$-OBDD into a regular OBDD. For this transformation all $\oplus$-nodes, starting from the root level are moved down towards the sink. As soon as at least one successor of a $\oplus$-node is a terminal node, it can be replaced by the result of the $\oplus$-operation. Thus, each $\oplus$-node that is moved down can be replaced and in the end the result will be a regular OBDD. The algorithm traverses the $\oplus$-OBDD levelwise starting from the root. In the traversal all $\oplus$-nodes of the current level that are encountered will be moved down by one level, before we proceed with the next level. If one of the new successors of the moved $\oplus$-node v is a terminal node, the swap-down procedure automatically applies possible reduction rules. A $\oplus$-nodes v can be replaced, if at least $k-1$ of its k successors are terminal nodes. All $\oplus$-nodes are replaced, if the algorithm has finished the processing of the last level of the $\oplus$-OBDD.
Note that the size of the decision diagram might grow exponentially while mov-

```
Input: ⊕-OBDD P_f
Output: OBDD P_f'

transform_to_OBDD(P_f) {
   // traversal
   for each variable table T_i, 1 ≤ i ≤ n do {
     for each node v ∈ T_i do {
       if ( v is ⊕-node ) {
         swap_down(v); // reductions are automatically applied
       }
     }
   }
   return(P_f);
}
```

Figure 5.32: Transformation of a ⊕-OBDD into a OBDD.

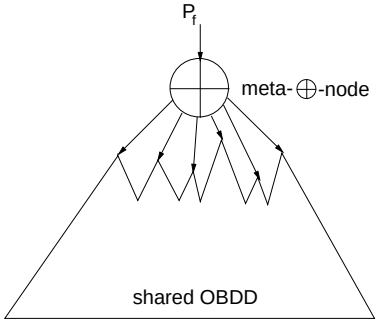


Figure 5.33: XOR-SOP Represented as ⊕-OBDD.

ing down all ⊕-nodes. See Fig. 5.32 for a sketch of the algorithm in pseudo code.
On the other hand, if we simply reverse the algorithm by moving up all ⊕-nodes of a ⊕-OBDD P_f to the root variable level, the result of this operation is equivalent to the representation of a XOR-SOP, i.e. a 3-level representation of the function f represented by P_f (see Fig. 5.33). This representation of a XOR-SOP can be used for further minimization, e.g. by applying the regular sifting algorithm to the branching nodes of all levels.

5.3 Dynamically Changing the Variable Order

In this section we are to investigate the effects of changing the variable order π for ⊕-OBDDs dynamically. Of course the improvement of the variable order to decrease ⊕-OBDD size is an NP-hard task. Thus, we are depending on heuristics, which are limiting the huge search space of $n!$ possible variable orders to a manageable amount of orders that are considerably simple to manage. But,

if we are to exchange adjacent variables x_i and x_{i+1}, which should be the basic task for dynamic variable order improvement, we have to take care of meta-$\oplus$-nodes, which might be located between these variables. As possible solutions for this problem there are two approaches to consider:

(1) Move the meta-$\oplus$-nodes out of the way, so that there are none of them between the two variables that are to be exchanged, and apply a regular variable exchange procedure then.

(2) Perform the variable exchange operation around meta-$\oplus$-nodes that are possibly located between the two variables that are to be exchanged.

Based on these two strategies we are able to adapt already known optimization heuristics that have proven to be successful for OBDDs and apply them to $\oplus$-OBDDs.

5.3.1 Adapting OBDD Minimization Heuristics

If we are going to implement the approach listed in (1), several difficulties will occur. By moving the $\oplus$-nodes out of the way for enabling the regular OBDD variable exchange procedure, the $\oplus$-nodes possibly are moved away from a previously well suited position and the size of the $\oplus$-OBDD might grow dramatically, if this procedure is carried out for an entire level in the $\oplus$-OBDD. Thus, after the variable exchange has been performed successfully, the transferred $\oplus$-nodes again have to be moved in a potentially better position, what can be achieved by applying the jiggle-algorithm to the involved levels. But, the two drawbacks of this approach remain:

- the algorithm is rather time expensive, because for each single variable exchange it requires several computation and optimization steps, and
- the achieved peak sizes during the computation might prevent the algorithm from its successful application, because given resource limits might be exceeded.

Another drawback in general lies in the fact that the variable exchange according to method (1) is not symmetric. This is caused by possible reductions and merges that are not reversible again, as it has been pointed out in the previous section for the jiggle algorithm. Thus, the adaption of the sifting algorithm for $\oplus$-OBDDs, where for each position of a variable within a given variable order the achieved BDD size is memorized and finally, the variable is placed in the level with the smallest size, will fail if we chose approach (1), because for $\oplus$-OBDD sifting it might be impossible to achieve the memorized best size ever again.

5.3.2 The Swap In Place Algorithm

In difference to the technique presented in the last section, we have the possibility to keep the $\oplus$-nodes in the place where they are, while exchanging the

branching nodes that are connected to them and that are adjacent in the variable order. An efficient implementation of this algorithm is possible, because we have introduced the concept of meta-$\oplus$-nodes. The usage of meta-$\oplus$-nodes guarantees that between any two branching nodes there must be at most one single meta-$\oplus$-node. Because of that rule, we have achieved a standardized setting for adapting the variable exchange procedure of OBDDs for $\oplus$-OBDDs: Let the variables to be exchanged be x_i and x_{i+1}. The $\oplus$-OBDD under consideration has the root node v labeled with x_i, and the two successors v_{x_i} and $v_{\overline{x_i}}$. If v_{x_i} and $v_{\overline{x_i}}$ are branching nodes, then the regular variable exchange procedure for OBDDs takes place.
Now, let us consider that v_{x_i} is a $\oplus$-node that is labeled with a reference to the variable x_{i+1}, i.e. the successor of v_{x_i} that is first w.r.t. the variable order is labeled with x_{i+1}. Let n_1 be the number of successors of v_{x_i}, and let $v_{x_i 1}, \ldots, v_{x_i n_1}$ be the successors of v_{x_i}.
For the exchange, v is simply relabeled with the variable x_{i+1} and connected to two successing $\oplus$-nodes $v_{x_{i+1}}$ and $v_{\overline{x_{i+1}}}$ that have to be newly created. The number of successors of $v_{x_{i+1}}$ and $v_{\overline{x_{i+1}}}$ is computed to be the maximum of the number of successors of v_{x_i} and $v_{\overline{x_i}}$ that we will denote as $n_{max} = \max(n_1, n_2)$. The successor nodes of $v_{x_{i+1}}$ and $v_{\overline{x_{i+1}}}$ that are denoted with $v^1_{x_{i+1}}, \ldots, v^{n_{max}}_{x_{i+1}}$ and $v^1_{\overline{x_{i+1}}}, \ldots, v^{n_{max}}_{\overline{x_{i+1}}}$, will be new branching nodes that are labeled with the variable x_i.
A successor node $v^i_{x_{i+1}}$, $i \leq \min(n_1, n_2)$, if the original successor $v^i_{x_i}$ is labeled with x_{i+1}, will have $(v^i_{x_i})_{x_{i+1}}$ as a left successor and $(v^i_{\overline{x_i}})_{x_{i+1}}$ as a right successor. Otherwise, if the original successor $v^i_{x_i}$ is not labeled with x_{i+1}, then the left successor will be $v^i_{x_i}$ itself. The same holds for $v^i_{\overline{x_i}}$, respectively.
For $n_1 < i \leq n_{max}$ the left successor will be the 0-sink. Otherwise, if $n_1 > n_2$, then, for $n_2 < i \leq n_{max}$ the right successor will be the 0-sink.
If one of the two nodes v_{x_i} and $v_{\overline{x_i}}$ is a branching node, while the other is a meta-$\oplus$-node, we proceed in the same way, while considering the number of successors of the branching node as $n_i = 2$. See Fig. 5.34 and Fig. 5.35 for two illustrating examples and Fig. 5.36 for a simple outline of the algorithm in pseudo code.
The algorithm given in Fig. 5.36 only gives a short outline of the idea behind the swap algorithm. In fact, the *swap-in-place* algorithm for $\oplus$-OBDDs differs from the variable exchange algorithm for OBDDs in the sense that several special cases for the implementation have to be considered. For the OBDD variable exchange algorithm only the cases listed in Fig. 5.37 have to be considered, where a branching node v_{x_i} labeled with the variable x_i has to be exchanged with its successor nodes that are labeled with the variable x_{i+1}. Either both of v_{x_i}'s successors are labeled with x_{i+1} (a), only one of the successors might be labeled with x_{i+1} (b), or even none (c). Additionally, there might be nodes labeled with x_{i+1} that don't have a predecessor in the level labeled with x_i (d). These four cases are treated in in the following way:

(a) This is the default case. The root node v has to be relabeled with x_{i+1}, two new successor nodes v'_{x_i}, $v'_{\overline{x_i}}$ of v have to be created or retrieved from the hash table labeled with x_i, with v'_{x_i} having successors f_1, f_3, and $v'_{\overline{x_i}}$

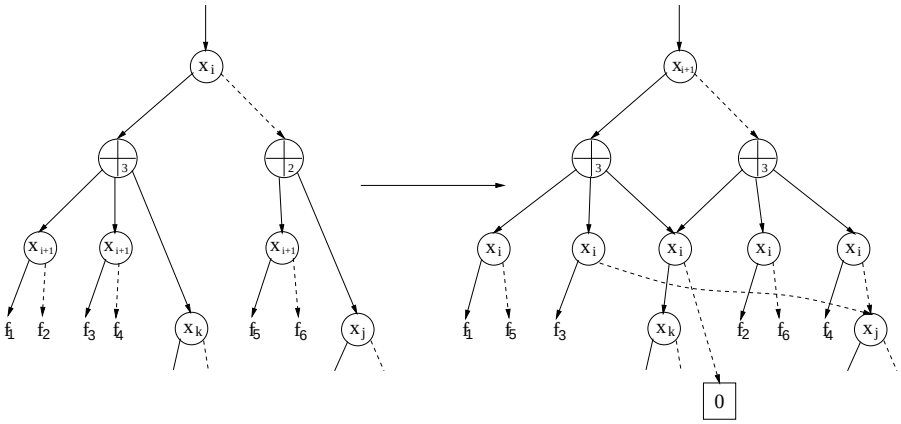


Figure 5.34: Example (1) for Swap-In-Place Operation for $\oplus$-OBDDs.

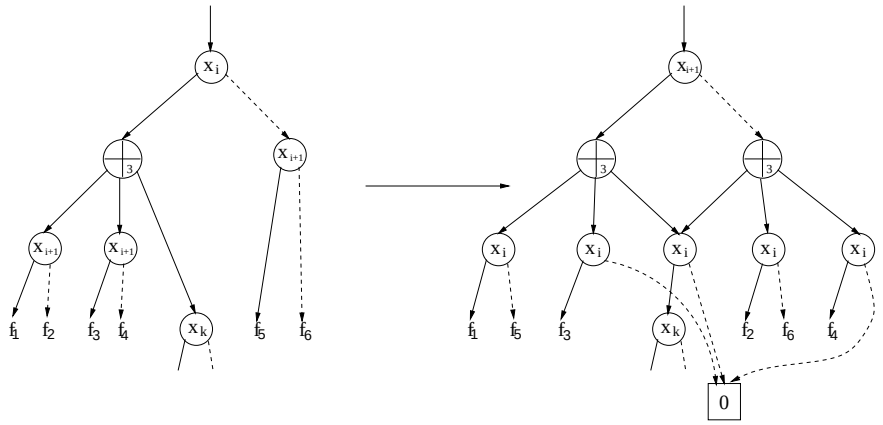


Figure 5.35: Example (2) for Swap-In-Place Operation for $\oplus$-OBDDs.

Input: $\oplus$-OBDD P, with root node v, $l(v) = x_i$.
Output: $\oplus$-OBDD P' with variables x_i and x_{i+1} exchanged.

```
swap_in_place(v) {
  v_{x_i} = v.1-succ; v_{¬x_i} = v.0-succ;
  if ( v_{x_i} and v_{¬x_i} are both branching nodes) {
    v = obdd_swap_regular(v);
  } else {
    n_max = max(v_{x_i}.nsucc, v_{¬x_i}.nsucc);
    prepare empty ⊕-nodes v_{x_{i+1}}, v_{¬x_{i+1}}
    for k = 1 to n_max do {
      if ( k < v_{x_i}.nsucc ) {
        if ( l(v^k_{x_i}) == x_{i+1} ) {
          v^k_{x_{i+1}}.1-succ = v^k_{x_i}.1-succ;
          v^k_{¬x_{i+1}}.1-succ = v^k_{x_i}.0-succ;
        } else {
          v^k_{x_{i+1}}.1-succ = v^k_{x_i};
          v^k_{¬x_{i+1}}.1-succ = v^k_{x_i};
        }
      } else {
        v^k_{x_{i+1}}.1-succ = 0-sink;
        v^k_{¬x_{i+1}}.1-succ = 0-sink;
      }
      if ( k < v_{¬x_i}.nsucc ) {
        if ( l(v^k_{¬x_i}) == x_{i+1} ) {
          v^k_{x_{i+1}}.0-succ = v^k_{¬x_i}.1-succ;
          v^k_{¬x_{i+1}}.0-succ = v^k_{¬x_i}.0-succ;
        } else {
          v^k_{x_{i+1}}.0-succ = v^k_{¬x_i};
          v^k_{¬x_{i+1}}.0-succ = v^k_{¬x_i};
        }
      } else {
        v^k_{x_{i+1}}.0-succ = 0-sink;
        v^k_{¬x_{i+1}}.0-succ = 0-sink;
      }
      v^k_{x_{i+1}} = create_new_or_find ( v^k_{x_{i+1}} );
      v^k_{¬x_{i+1}} = create_new_or_find ( v^k_{¬x_{i+1}} );
      v_{x_{i+1}} = insert_in_successor_list ( v^k_{x_{i+1}} );
      v_{¬x_{i+1}} = insert_in_successor_list ( v^k_{¬x_{i+1}} );
    }
    v.1-succ = v_{x_{i+1}};
    v.0-succ = v_{¬x_{i+1}};
    l(v) = x_{i+1};
  }
  return(v);
}
```

Figure 5.36: Sketch of the Swap-in-Place Algorithm for $\oplus$-OBDD in Pseudo Code.

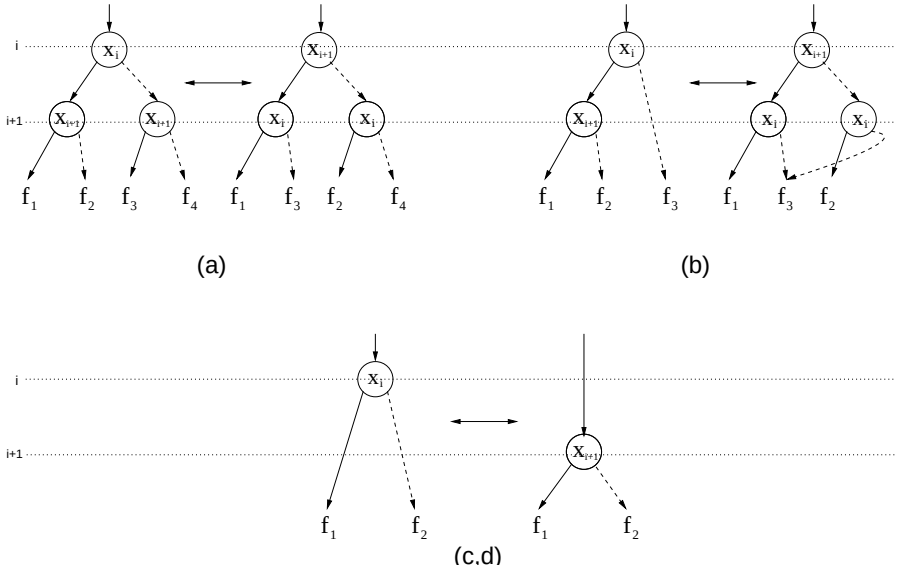


Figure 5.37: Implementation of Variable Exchange for OBDDs.

having successors f_2, f_4. The reference counter of the old successors v_{x_i} and $v_{\overline{x_i}}$ is decremented.

(b) The two cases for v having only one successor labeled with x_{i+1} are symmetrical. W.l.o.g. let v_{x_i} be labeled with x_{i+1}. Also two new successor nodes v'_{x_i}, $v'_{\overline{x_i}}$ are created or retrieved from the hash table labeled with x_i, with v'_{x_i} having successors f_1, f_3, and $v'_{\overline{x_i}}$ having successors f_2, f_3.

(c,d) If none of the successors of v is labeled with x_{i+1} then v is simply transfered to the hash table of the next variable level. The same holds for case (d), where v is labeled with x_{i+1} without having a predecessor in level i. There, v is simply transferred to the hash table of level i

Note that already transferred nodes or newly created nodes have to be marked such that they are not affected by the ongoing exchange procedure. This can be achieved by either using spare bits of the data structure for setting marks or by using temporary hash tables, which will substitute the old hash table after the variable exchange procedure has finished.
Now, for the *swap-in-place* algorithm for $\oplus$-OBDDs, even more different cases have to be considered. Additionally to the cases listed for OBDDs, it has to be taken under consideration that the successing nodes of v_{x_i} might be $\oplus$-nodes.

- In the same way as in the four cases above, both successing nodes can be labeled with x_{i+1}, while at the same time both successor nodes are $\oplus$-nodes (a1), only one of the successing nodes might be a $\oplus$-node (a2) or none (a).

- If only one successing node is labeled with x_{i+1}, this node might be a $\oplus$-node (b1) or a branching node (b).

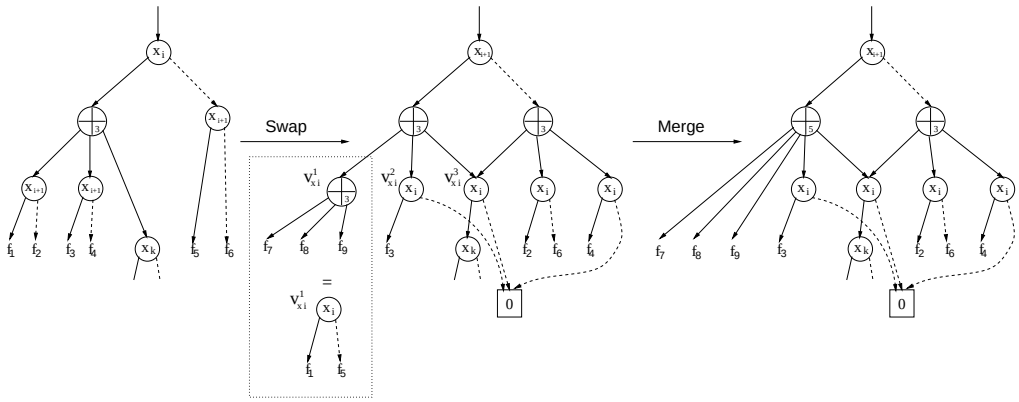


Figure 5.38: Merge During the Swap-in-Place Procedure.

- Then, of course none of the successing nodes of v_{x_i} might be labeled with x_{i+1} (c).
- Now, v_{x_i} itself might be a $\oplus$-node labeled with a reference to x_i, i.e. at least one of it's successors must be a node labeled with x_i. Then, we have to distinguish between the case that v_{x_i} also has a successor labeled with the adjacent variable x_{i+1} (d1) or not (d2).
- The last case denotes a $\oplus$-node $v_{x_{i+1}}$ that possesses a successor node labeled with x_{i+1}, but which does not have any predecessor labeled with x_i (e).

These five cases are treated in the following way:

(a) If none of the successing nodes is a $\oplus$-node, then the situation is equivalent to OBDDs and can be solved as (a) stated in the previous enumeration. If both successing nodes are $\oplus$-nodes, both labeled with x_{i+1} (a1), we have the standard situation as described in algorithm 5.36 (see also Fig. 5.34) Also, for the case that only one of the successing nodes is a $\oplus$-node, but both successors are labeled with x_{i+1} (a2) the case is similar to the previously described standard situation (see Fig. 5.35).

A special situation that has to be taken care of is the following:

While creating new successor nodes $v_{x_i}^k$, $1 \leq k \leq n_{max}$ it is possible that a node to be created does already exist in the hash table of x_i. But, according to our requirement concerning meta-$\oplus$-nodes, $v_{x_i}^k$ should be a branching node. If the already existing node $v_{x_i}^k$ from the hash table is a $\oplus$-node, we have to merge $v_{x_i}^k$ with it's predecessor meta-$\oplus$-node $v_{x_{i+1}}$ (see Fig. 5.38) according to our rule that is mandatory for efficient variable exchange for $\oplus$-OBDDs.

(b) In the case that only one of the successors of v is a $\oplus$-node labeled with a reference to x_{i+1} the processing does not differ from the previous case, but the following exception will become obvious, when reversing the operation:

Consider the situation given in Fig. 5.39. In the $\oplus$-OBDD P the two variables x_i and x_{i+1} are to be exchanged, while the top node v has only one successor being a $\oplus$-node labeled with x_{i+1} (1). After the first variable exchange one of the two $\oplus$-OBDDs P_1' (2a) or P_2' (2b) is computed. Note that both $\oplus$-OBDDs represent the same Boolean function, but have a different structure. This is due to the fact that in the two cases, the order of the $\oplus$-node successors is changed (see dotted box in Fig. 5.39). Up to now, the order of $\oplus$-OBDD successors could be neglected for $\oplus$-OBDD manipulation. But, in that particular situation, there is an important difference between the two results of the backtransformation, when the *swap-in-place* algorithm is applied again to the two variables (see (3a) and (3b)). Again, both $\oplus$-OBDDs, P and P_2'' represent the same Boolean function. The 0-successor of the top node in P_2'' represents also the function f_4, as for P, but in our implementation, we don't have a chance to realize that equivalence in an efficient way. This is, because for P_2'', the $\oplus$-node depends on the variable x_{i+1}. In P, the 0-successor depends on a variable x_j, $j > i+1$, or it is simply a sink. Thus, both are residing in different hash tables and their equivalence will not be realized.

A situation like that can be avoided either by the application of the extended reduction rule for $\oplus$-OBDDs given in Section 4.3 for meta-$\oplus$-nodes. See Fig. 5.40 for an example of the application of the extended reduction rule for reducing P_2'' into P.

But, if we always apply this reduction rule, this also implies the possibility of new non symmetrical situations, because the extended reduction rule tries to find an optimal combination of all 0- and 1-successors of nodes in the successor list of a $\oplus$-node labeled with the same variable. The reduction also implies the usage of already computed nodes, and then, again merges of adjacent $\oplus$-nodes are possible. After the reduction and merging, the application of the *swap-in-place* algorithm is not capable to restore the original situation because of the given reasons.

Therefore, we have to restrict the application of reduction rules and the usage of already computed nodes, for keeping the variable exchange operation for $\oplus$-OBDDs symmetrical.

(c) This case is equivalent to item (c) in the previous list.

(d) Let's first consider case (d2). In the $\oplus$-OBDD P with root node v being a meta-$\oplus$-node, connected to some node v_i that is labeled with the variable x_i, none of v's successors is labeled with x_{i+1}. Here, the variable exchange only takes place between v_i and its successors (see Fig. 5.41). v itself changes its label, but can remain within the same unique table. Note that in general for a variable exchange in the case of $\oplus$-OBDDs as well as in the case of OBDDs only nodes that are labeled with the variable x_{i+1} have to be put into a new table. One possibility is to use a temporary table for this purpose, because already processed nodes should not be mixed up with nodes that are still to be processed. Another implementation, as e.g. in CUDD, first empties the table of variable x_i by transferring all

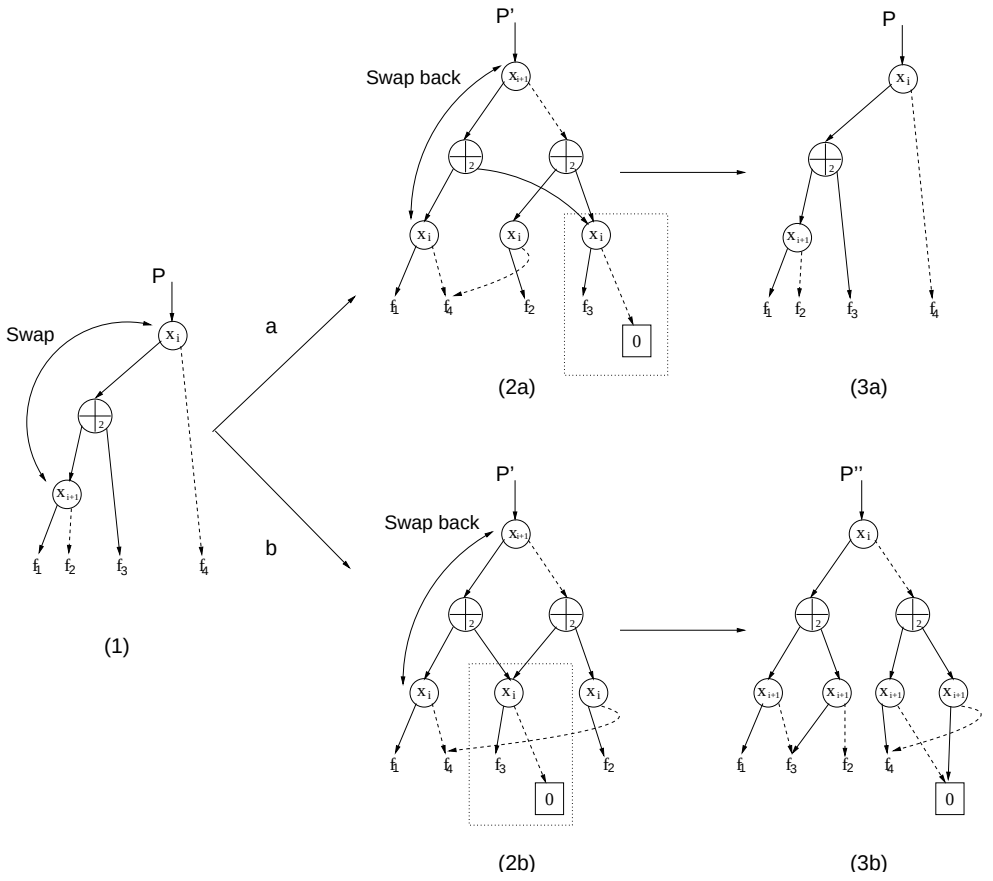


Figure 5.39: Non Symmetrical Situation for the Swap-in-Place Operation.

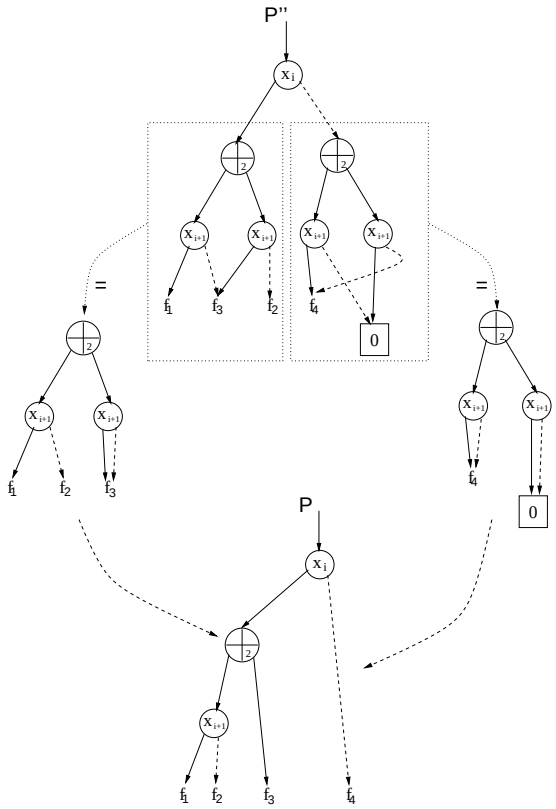


Figure 5.40: Keeping Symmetry by Adapting the Extended Reduction Rule to Meta-⊕-Nodes.

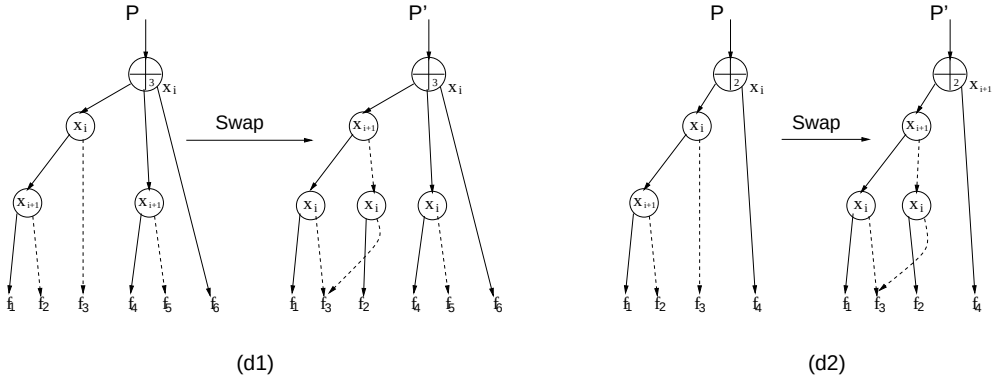


Figure 5.41: Considering Predecessor Meta-⊕-Nodes in the Swap-In-Place Algorithm (1).

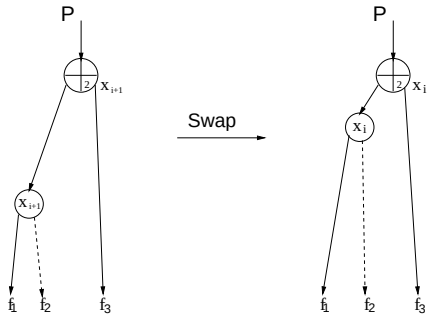


Figure 5.42: Considering Predecessor Meta-⊕-Nodes in the Swap-In-Place Algorithm (2).

nodes that have to be processed in some way into a separate temporary linked list.

For (d1), the situation is similar. v remains completely unchanged, because after the variable exchange it will still be connected to successors that are labeled with both x_i and x_{i+1}. The actual variable exchange takes only place for v's successors (see also Fig. 5.41).

(e) Here, v is labeled with a reference to x_{i+1}. None of v's successors is labeled with a reference to x_i. While all successors labeled with x_{i+1} are relabeled with x_i, also v has to be relabeled accordingly (see Fig. 5.42).

5.3.3 Adapted Sifting - A Heuristic for ⊕-OBDDs Minimization

In the previous section we introduced a basic operation for any dynamic variable reordering technique for ⊕-OBDDs - the *swap-in-place* operation. Now, for designing a suitable heuristic for ⊕-OBDD minimization that uses the swap-in-place operation, we have to consider the following two different cases:

(1) the swap-in-place operation is symmetric, i.e. applying this operation

twice to the same variable level must result in a $\oplus$-OBDD that is isomorphic to the original OBDD, or

(2) the swap-in-place operation is not symmetric.

For adapting the well known *sifting*-heuristic of OBDDs to $\oplus$-OBDDs, the basic operation must be symmetric. For, a distinct variable x_i, $1 \leq i \leq n$ is chosen from the given variable set of the OBDD P. Usually, the variable related to the level with the most nodes in it and that has not already been processed is chosen. With successive variable exchange operations, the variable x_i is transferred to every position within the variable order and for each position, the actual node count $P`_i$ is stored. In a final processing step, $x_i$ is transferred to level $k$, where $P`_k \leq P_i$, $\forall i$. This operation is repeated for every variable and finally, a local minimum for the size of P is achieved.
To implement the sifting heuristic for $\oplus$-OBDDs, the basic operation, the variable exchange must be symmetric. In the previous chapter, we realized that there are several situations, when the exchange does not meet this requirement:

(1) The reuse of already existing nodes during the exchange procedure. The exchange operation requires new nodes to be connected into the successor list of a $\oplus$-node between two adjacent variable layers. If the node to be reused from the hash table happens to be a $\oplus$-node and not a branching node, a merge operation is required, because no two $\oplus$-nodes are allowed to be adjacent. The merge alone is not reversible again. The situation gets worse, if reductions are also applied.

(2) Also applying the *extended* reduction rule for meta-$\oplus$-nodes leads to non reversible exchange operations. This reduction rule only takes place for branching nodes in the successor list of a $\oplus$-node that are labeled with the same variable. There, the 1- and 0-successors can be combined in the way that the least number of nodes is required. Of course, the result is functionally equivalent. But, if we try to reverse the variable exchange operation, we will gain a structurally different $\oplus$-OBDD.

If we keep all possible reduction and merging rules, we have to think of a different type of minimization heuristic. This can be a greedy heuristic that at least only memorized the entire affected node levels before and after the exchange operation. If there is some gain in size achieved by the swap operation, we keep the swap, otherwise we switch back to the memorized starting situation and try a swap in the other direction. After all possible minimizing swap operations for a single variable, we switch to the next variable in order and continue.
But, in this work we put the emphasis on finding a way of how to find a way to adapt the regular sifting algorithm, because this algorithm has proved to be a simple and one of the best heuristics for OBDD minimization.
The rule to keep the *swap-in-place* operation symmetric is quite simple:

(1) Renounce the application of the extended reduction rule for meta-$\oplus$-nodes, and

(2) Don't use existing $\oplus$-nodes from the hash table, when branching nodes are required to maintain the meta-$\oplus$-node rule.

But now, to proceed with the sifting heuristic, we have to consider the following requirements: if (1) is maintained and the extended reduction rule is not applied, a potentially smaller $\oplus$-OBDD for the represented Boolean function and for the same variable order might exist. This is an important issue, since, for the sifting heuristic, it is important to place the variable under consideration to the best place within the variable order, i.e. the place where the size of the entire $\oplus$-OBDD is the smallest for that particular variable. If we don't apply the extended reduction, we don't know, whether the actual position of the sifted variable is the best or not. To prevent that problem, we must keep track of potential reductions that might take place. For every hash table h_{x_i} in the $\oplus$-OBDD P, we maintain a counter r_{x_i} that stores the number of possible reductions taking place during a variable exchange operation. To calculate the actual node count P_{real}, the number of possibly reduced nodes is subtracted from the number of counted nodes $|P|$: $P_{real} = |P| - r_{x_i}$. Note that the hash table associated with the variable x_i keeps this association, while it switches the level in the $\oplus$-OBDD. To take into account all possible reductions, the extended reduction rule for $\oplus$-OBDDs is simulated for every node in the level and a separate data structure keeps track of the actual reference count of each node. Additionally, duplicate nodes that have been created in accordance to rule (2) do not have to be counted. First, we must take care that duplicate nodes, usually a branching node v_b and a $\oplus$-node $v_\oplus$, both labeled with the same variable x_i, are managed correctly. Both nodes reside in the same hash table h_{x_i} and occupy the same slot of the table. The hash table is organized to maintain ordered collision lists. Thus, both nodes, v_b and $v_\oplus$, are adjacent in the collision list, while one of them will be marked with a temporary single bit flag as being a duplicate. Note that e.g. the *next*-pointer of the node, connecting the node with its successor in the collision list of the hash table is also an address pointer and therefore, its least significant bit can also be used as a flag bit. In addition to that, also a counter for duplicate nodes d_{x_i} has to be maintained to compute the real node count for the sifting heuristic. Thus, the real size of P is computed $P_{real} = |P| - (r_{x_i} + d_{x_i})$.
Now, for each position of the variable x_i in the variable order, we keep track of the real size of the $\oplus$-OBDD P. x_i will be sifted into the position, where P_{real} was minimal.
Once x_i is in the correct position, a complete reduction step has to be performed. First, all duplicate nodes will be eliminated. This can be achieved rather simple by traversing P and redirecting all successor pointers of a node that are referencing a duplicate node to the node next in the collision list. The reference counter of the duplicate nodes is decremented and they will be removed in a final garbage collection. Next, starting bottom up, all possible applications of the extended reduction rule are performed for meta-$\oplus$-nodes in hash tables h_{x_i}, where $r_{x_i} > 0$. Now, P is completely reduced and the next variable can be processed. After all variables have been processed, the sifting algorithm is finished. See Fig. 5.43 for an outline of the sifting algorithm for

```
Input: ⊕-OBDD P_f, with variable order Π, and growth factor γ.
Output: ⊕-OBDD P'_f, P'_f ≤ P_f with variable order Π'.

mod2_sifting (P, γ) {
   create ordered list of variables x_i, 1 ≤ i ≤ n;
   foreach variable x_i {
    repeat {
     move x_i through all levels j, 1 ≤ j ≤ n, while
     storing |P_j|, the size of P with x_i in level j
     using the symmetric swap_in_place procedure;
    }
    until (|P_j| > γ · |P| or all levels j have been accessed)
    target = level j with |P_j| = min(P_j), 1 ≤ i ≤ n;
    move x_i to level target;
    remove all duplicate nodes in P;
    do complete extended reduction of P;
   }
   return(P);
}
```

Figure 5.43: Outline of the Sifting Algorithm for ⊕-OBDDs in Pseudo Code.

⊕-OBDDs in pseudo code.

To limit the growth of the ⊕-OBDD during the sifting procedure, a growth factor $\gamma \in \mathbb{R}$ similar as in the case for OBDDs is introduced, preventing the sifting algorithm to proceed beyond a point, where no improvement seems to be much likely. In practice, a growth factor of $\gamma = 1.2$, as it is implemented as the default growth factor in the CUDD package, has shown to be rather efficient. Thus, we also chose $\gamma = 1.2$.

The major difference in the sifting procedure between OBDDs and ⊕-OBDDs is the lack of symmetry in the basic variable exchange procedure. For OBDDs, this swap is always reversible including all possible reductions to be carried out. This is due to the canonicity property of the OBDD and because the reductions always lead to a canonical form of the function to be represented as an OBDD. For ⊕-OBDDs, for the same Boolean function, there might exist several structurally different ⊕-OBDDs and the possible reduction rules don't guarantee a canonical form. By preventing certain reductions for ⊕-OBDDs during the variable swap, we keep the swap operation to a fixed regular form, allowing possible duplicate nodes and unreduced meta-⊕-nodes (unreduced w.r.t. the extended reduction rule). These prerequisites have to be taken into account and after successfully processing a single variable, we have to clean up redundant nodes and unreduced subgraphs.

To apply the sifting heuristic efficiently in synthesis, it makes only sense to apply the variable reordering algorithm already dynamically during the synthesis process to prevent the ⊕-OBDD from becoming too large. If the heuristic is only applied afterwards, after synthesis has finished, the ⊕-OBDD might already have become too large to be represent with the available resources. Thus,

during the synthesis process, the construction of the $\oplus$-OBDD P is stopped, after the size of P has exceeded a certain threshold bound α_0. If $|P| > \alpha_0$, the sifting routine is called for the function that is already represented. Afterwards, synthesis continues, until the next threshold value $\alpha_1 = \gamma \cdot \alpha_0$ is exceeded. As for the dynamic jiggle algorithm, we chose $\gamma = 2$. Thus, the threshold value α_i, $i \geq 0$ doubles after each iteration. When the synthesis of the $\oplus$-OBDD has finished, a final reorder process might take place to optimize the already found variable order.

For improving the $\oplus$-OBDD size, also the Jiggle algorithm can be called either during or after synthesis. But, we should limit the number of its application, because as shown in the previous chapter, this algorithm is rather costly and has not necessarily a significant positive impact on $\oplus$-OBDD size.

To show the efficiency of the heuristic, again we performed symbolic simulation of the same benchmark circuit set. First, we compared OBDD sizes against $\oplus$-OBDD sizes, both achieved with dynamic reordering enabled and *sifting* as the designated reordering algorithm. The threshold value, when to start variable reordering for $\oplus$-OBDDs, was adopted from CUDD and set to $\alpha_o =$ 4004 nodes. The limiting growth factor for both cases was set to $\gamma = 1.2$. In the first round of experiments (1), we compared only sizes achieved by the sifting heuristic, while in the second round (2), additionally the jiggle heuristic for $\oplus$-OBDDs was applied to gain a further improvement of $\oplus$-OBDD size. Additionally, to demonstrate the optimization potential of the sifting algorithm, for a comparison we also compared the size of $\oplus$-OBDDs with the variable order given by the circuit and the size of the $\oplus$-OBDD for the same circuit with the variable order achieved with the sifting algorithm (3). To get a suitable number of $\oplus$-nodes into the $\oplus$-OBDD, the greedy heuristic for deciding, when to use *standard ITE* or *pDE*, was applied.

See Table 5.11 for a an overview of some selected results for (1), (2), and (3). For the results of the complete benchmark circuit set see Tables A.20 and A.21 in the Appendix.

In the first column the circuit name is given, while in the second column the size of the OBDD achieved with the application of dynamic variable reordering and the sifting heuristic is listed. The third column gives the size of the $\oplus$-OBDD and the standard variable order given within the circuit description. Synthesis in all cases for $\oplus$-OBDDs has been performed with application of the simple greedy heuristic in combination with the apply-$\oplus$ algorithm based on the pDE-decomposition. Column four shows the $\oplus$-OBDD sizes for the application of dynamic reordering with the extended sifting heuristic for $\oplus$-OBDDs, while column five lists additionally the sizes for $\oplus$-OBDDs when dynamic $\oplus$-node relocation with the dynamic jiggle algorithm is computed.

In the upper part of the tables in the appendix, the sizes for all circuits is given that have also been computed up to now within the given resource limitations. Note that with the variable reordering heuristic enabled, it is possible to compute additional circuits that are listed in the lower part of the tables.

Next, we proceed with the discussion of the results for the single experiments:

(1) Here, a comparison of OBDD size and $\oplus$-OBDD size for synthesis with

Circuit	OBDD-size	⊕-OBDD size [Bytes]		
	Sifting	**pDE-meta**	**Sifting**	**+ Dynamic Jiggle**
pair	128664	3409992	106776	106376
my_adder	4716	18873336	5868	5100
i7	14148	31172	20800	13928
i2	7380	10880	7344	7344
bw6x6	28980	95832	67940	67628
alu2	5832	9856	5768	5760
C499	958464	670188	196708	187892
C432	43560	184040	37160	36792
C1355	1064232	670024	1065340	1064196
comp	4608	27142104	3928	3796
i10	2446956	-	1906792	1904572
mm30a	3621276	-	3010564	3009244
C7552	296674	-	254196	254196
sbc	38052	135332	38556	35672
s820	8172	16532	7044	6264
s713	22644	115428	18408	17888
s641	22644	115360	16044	14660
s635	4716	64692	10364	3356
s444	5796	11136	1588	1340
s1269	78804	1101680	79140	76328
mm9b	90936	18157936	74108	73292
mm9a	72828	14583716	57264	56472
mult16a	7380	23852212	8032	8032
s838.1	6444	-	6516	5984
s3384	32616	-	40744	32432
Σ	**13.701.388**	**-**	**12.700.536**	**12.535.556**
	100%		**92.7%**	**91.5%**

Table 5.11: Comparison of OBDD and ⊕-OBDD size for the Sifting Heuristic.

Circuit	OBDD	⊕-OBDD		
	Sifting	**pDE-meta**	**Sifting**	**+ Dynamic Jiggle**
$\Sigma_{\textbf{previous}}$	54.7	2687	526.3	677.8
$\text{norm}_{\text{previous}}$	1.0	49.1	9.6	12.4
Σ	620	-	8308	12961
norm	1.0	-	13.4	20.9

Table 5.12: Runtime Requirements for the Sifting Heuristic for OBDDs and ⊕-OBDDs.

dynamic sifting enabled is conducted. If we compare the overall achieved sizes, the results for OBDDs and $\oplus$-OBDDs are quite similar, while for $\oplus$-OBDDs the overall achieved size with 92.6% of the OBDD-size is a little bit smaller. The important thing to mention is that now, with the sifting heuristic enabled, it is possible to perform synthesis of 14 circuits of our benchmark set that were not computable within the given resource limitations with the previous methods. If we distinguish the sizes achieved for sequential and combinatorial circuits, with 92.6% of the OBDD-size, $\oplus$-OBDDs perform better for combinatorial circuits compared to sequential circuits, where they come up to 96.3% percent in size. With a few exceptions, the results achieved for $\oplus$-OBDDs are always smaller or at least comparable to OBDDs. These exceptions include circuits that also previously have shown not to benefit too much from the introduction of $\oplus$-nodes. The worst case of these examples is *bw6x6*, where $\oplus$-OBDD size is about 233% the size of the OBDD. But, there are also examples of circuits that really do benefit from the introduction of $\oplus$-nodes, as e.g. *C499* or *s444*. There the $\oplus$-OBDD size could be reduced up to 20% (27%) of the OBDD size.

All in all, the impact of $\oplus$-OBDD nodes is not as obvious as for not optimized variable orders. The gain in size is less than 10%, if measured over all circuits of the benchmark in the average. This shows that the percentage of circuits that really do benefit from the use of $\oplus$-OBDDs, at least in this particular benchmark set, is small.

(2) Here, additionally, for $\oplus$-OBDD synthesis with enabled sifting heuristic, also the dynamic jiggle heuristic was applied during construction and compared with standard OBDD sifting. The overall improvement for all circuits that could be achieved additionally to the sifting heuristic comes only up to 1%. Thus, in the average, for most circuits the application of dynamic jiggle during synthesis does not really fall into account. But, there are some circuits, where huge improvements could be achieved, e.g. *s635* that could be reduced from 219% or the OBDD size to only 71%, or *i7*, where a reduction from 147% to 98% was possible. But for the major part of the circuits, only a small reduction could be achieved. If we take the additional amount of time required for the jiggle algorithm into account, its application is not really efficient.

(3) Finally, a comparison of $\oplus$-OBDD size with and without application of the sifting algorithm was performed. The results show the huge impact of the variable order on the $\oplus$-OBDD size. First at all, there are 14 circuits that could not be computed with the variable order given within the circuit file. The sifting heuristic was able to keep the computation within the given resource limitations. The reduction in size for the sifting algorithm is tremendous. The original $\oplus$-OBDD is about 40 times larger than the size achieved with the sifting heuristic.

After having shown, that the sifting heuristic is able to reduce $\oplus$-OBDD size in a similar way that for OBDDs, we have to evaluate its efficiency in terms

of computation time also. See Table 5.12 for the overall required computation time for the application of the sifting heuristic. As in the previous table, the second column lists the required time for OBDDs, while column 3, 4, and 5 lists computation time for $\oplus$-OBDDs without sifting, with sifting, and with additional application of the dynamic jiggle heuristic. The first row gives the achieved results for those circuits only that were also able to be computed previously without the sifting heuristic to have a suitable comparison. The second row contains the results for all circuits in seconds, and the last row shows the results normalized in relation to the time required by OBDDs and the CUDD package.

Let us first compare the results for those circuits that have been already computable with the previous method that are in the upper part of the table. Also with dynamic variable reordering enabled, the CUDD package can defend its leading position, but compared to the runtime requirements for the static variable order given by the circuits, the $\oplus$-OBDD sifting is capable of coming up closer. Although now with sifting enabled, the runtime is up to ten times higher compared to OBDDs, it is much faster compared to the static variable order synthesis for $\oplus$-OBDDs. This speed up is achieved by avoiding memory peak sizes and avoiding the maintenance of huge node counts within the $\oplus$-OBDD package. If we take the other circuits that could not be computed previously also into account, we notice a slow down compared to CUDD and the factor of being slower is now about 13 times. This additional slowdown can be explained easily. As we have shown before, the main reason for the $\oplus$-OBDD package for being so slow are the signature based equivalence test, the more complex cofactor creation algorithm, and the not so sophisticated memory management compared to CUDD. If now, larger circuits are processed the influence of the memory management becomes much higher. On the other hand, the sifting algorithm for $\oplus$-OBDDs is more complicated compared to OBDD sifting. We have to apply additional reduction steps before we can switch to the next variable to be repositioned. Taking all these factors into account, this results in the important slow down compared to OBDDs.

As for many other data structures too, smaller representation size can often only be achieved by an increase of the complexity of the management algorithms related to that data structure. The same holds for $\oplus$-OBDDs. Although we have shown that there is the possibility of exponential gaps between OBDD and $\oplus$-OBDD representation size, it does not really have the same effect for the circuits of our benchmark set. There, the decrease in size in the average comes to not even 10% for the processing, when the optimization heuristic is enabled. Unfortunately, this small win must be paid with a much larger loss in runtime.

5.3.4 Conclusions

Do $\oplus$-OBDDs have any significant influence in practice, after the results of this work being published? Certainly, for applications, where we are able to perform dynamic reordering with OBDDs and where we have designs that can be represented with OBDDs within the given resource limitations, there, most likely they will not have any major relevance.

But, there are also cases in practice, where dynamic reordering is not suitable, because the designs simply are too huge. There, a more or less well suited static variable order is computed from the given circuit design and then, the circuit is to be represented as an OBDD following that static variable order. There, where the representation size is of upmost importance, there is the chance for $\oplus$-OBDDs also to be useful in practical applications.
And we should not forget that the achieved results are only the results on a more or less small benchmark set and not necessarily representative for all possible designs. It has been shown that there are exponential gaps between the representation sizes of OBDDs and $\oplus$-OBDDs and certainly - as a few of the given benchmark circuits also show - there must be designs, where this advantage can be rather useful.

Chapter 6

Extension of $\oplus$-OBDDs to the Discrete Domain

In this chapter we show how to extend the $\oplus$-OBDD data structure from its original binary domain to an arbitrary discrete domain. We distinguish bit-level decision diagrams representing functions with Boolean outputs and word-level decision diagrams representing functions whose outputs can be arbitrary integers. We do not consider extensions of BDDs that are based on the arithmetization of the decomposition rules like Binary Momentum Diagrams (BMDs) [BC95] or the introduction of multiplicative or additive edge values as for Edge Valued Binary Decision Diagrams (EVBDDs) [LS92, LPV94] or Edge Valued Binary Momentum Diagrams (*BMDs). The focus of this chapter lies on Multiple Valued Decision Diagrams (MDDs), where each decision node has more than just two outgoing edges and is extending OBDDs to a finite domain. As for OBDDs, also word-level DDs might benefit from the introduction of operator nodes. For this reason we are adapting the concept of MDDs to Mod-p Decision Diagrams (Mod-p-DDs) that are a generalization of $\oplus$-OBDDs.

6.1 Multiple Valued Decision Diagrams

In difference to Multi Terminal Binary Decision Diagrams (MTBDDs) [CFM+93] that are also known as Algebraic Decision Diagrams (ADDs) [BFG+97], which are representing functions of type $f : \{0,1\}^n \rightarrow M$, $M \subset \mathbb{Z}$ and allow a compact representation of sparse matrices as they are used in the verification of probabilistic systems, we consider another type of OBDD extension for the representation of multiple-valued logical functions of type $f : M^n \rightarrow M$, $M = \{0, 1, \ldots, p-1\}$, $p \in \mathbb{N}$. In order to achieve this, multiple-valued decision variables have to be introduced that are able to take an arbitrary number p ($p > 2$) of different values. A decision diagram that is constructed out of nodes that are representing multiple-valued decision variables is called *Multiple Valued Decision Diagram* (MDD) [SKM+90].

Definition 6.1 *A Multiple Valued Decision Diagram on the variable set* $X_n = \{x_1, \ldots, x_n\}$, *where* x_i *might take values from* $M = \{0, 1, \ldots, p-1\}$, $p \in \mathbb{N}$ *is*

representing a multi valued function $f : M^n \to M$. Every non terminal node of a MDD is labeled with a variable $x_i \in X_n$ and has p outgoing edges. An input $a = \{a_1, \ldots, a_n\}$, $a_i \in M$ activates a path starting in the source, choosing the a_i-edges at nodes labeled with x_i, leading to one of the p terminal nodes $t_j \in \{t_0, \ldots, t_{p-1}\}$ that are labeled with one of the constants $c_j \in M$.

Note that in general it is possible to simulate a node v of a MDD P with p outgoing edges by a binary decision tree of depth $d = \lceil \log p \rceil$ with $2^d - 1$ Boolean branching nodes and 2^d leaves. All Boolean variables that are constructed in this way test only a fraction of a multi valued variable. Because of this binary encoding, there are more possible variable orders and thus, there is the potential of constructing a smaller data structure by using this transformation. But, if the considered problem definition is given in terms of multi valued variables, MDDs are rather useful.
In an arbitrary MDD P for each node v labeled with the multi valued variable $x_i \in X_n$ the generalized Boole-/Shannon decomposition [MW86] is applied as follows:

$$f_v(x_1, \ldots, x_n) = \overset{0}{x_i} \cdot f_{x_i=0} + \overset{1}{x_i} \cdot f_{x_i=1} + \ldots + \overset{m-1}{x_i} \cdot f_{x_i=m-1},$$

where $f_{x_i=j}$ are cofactors of f, and '+','·' denote the multi valued operations MAX and MIN, correspondently. $\overset{i}{x}$ is the unary operation *literal*, which is the multi valued generalization of the Boolean negation, and is defined by

$$\overset{i}{x} = \begin{cases} p-1 & \text{if } x = i \\ 0 & \text{otherwise.} \end{cases}$$

The *cofactor* of a multiple valued function $f : M^n \to M$ with respect to a variable x_i is the function resulting when x_i is replaced by a constant value $b \in M$ and is denoted by $f|_{x_i=b}$ or simply $f_{x_i^b}$:

$$f_{x_i^b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)$$

Definition 6.2 *Let $X = \{x_0, \ldots, x_{n-1}\}$ be a set of n multi valued variables and let $\pi : X \to \{0, 1, 2, \ldots, |X| = n-1\}$ be a bijective mapping of the variable indices. An* **Ordered Multiple-Valued Decision Diagram (OMDD)** *is a MDD, where the order in which the variables occur is consistent with π, i.e. if there is an edge leading from a node labeled with x_i to a node labeled with x_j, then $\pi(x_i) < \pi(x_j)$ must hold. On each path from the root to a terminal node all variables must occur at most once.*

The concept of reduction that has been introduced with OBDDs can directly be adapted for OMDDs:

Deletion rule: (*simple reduction*) If all outgoing edges of a node v lead to the same node w, then eliminate v and redirect all incoming edges from v to w.

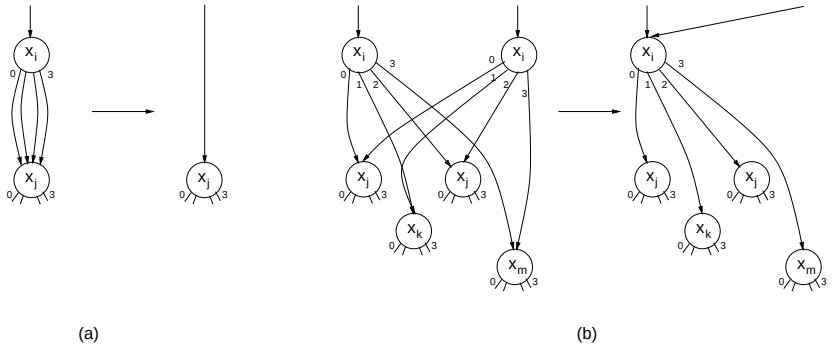


Figure 6.1: Reduction Rules for OMDDs.

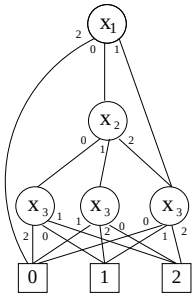


Figure 6.2: Example of a 3-Valued OMDD.

Merging rule: (*algebraic reduction*) If the nodes v and v' are labeled with the same m-valued variable x_i, and for all of their successors it holds that $v_{x=k} = v'_{x=k}$, $\forall k \in \{0, \ldots p-1\}$, then one of the nodes can be eliminated while redirecting all incoming edges to the other node.

See Fig. 6.1 for an illustration in case of a $p = 4$ valued OMDD. If none of the above reduction rules can be applied to the OMDD P_f, it is called *reduced*. Reduced OMDDs are a canonical representation for multi valued functions, as OBDDs are for Boolean functions. For multiple output functions $f : M^n \to M^r$, $r \geq 1$, we can represent these functions by a single graph with multiple root nodes, a structure called *shared OMDD*.
See Fig. 6.2 for an example of an OMDD P_f that is representing a function $f : M^3 \to M$ with $M = \{0, 1, 2\}$. f is defined by the following value table:

$x_3 \backslash x_1 x_2$	00	01	02	12	11	10	20	21	22
0	1	2	0	0	0	0	0	0	0
1	2	0	1	1	1	1	0	0	0
2	0	1	2	2	2	2	0	0	0

For the synthesis of OMDDs, the well known ITE algorithm for OBDDs has been generalized to the multi valued **CASE**-algorithm [Mil93] in the following way: Let f and $g_0, \ldots, g_{p-1}$ be $p+1$ p-valued functions. The $CASE$-operator

is defined as

$$case(f, g_0, \ldots, g_{p-1}) = \overset{0}{f} \cdot g_0 + \overset{1}{f} \cdot g_1 + \cdots + \overset{p-1}{f} \cdot g_{p-1}$$

As for the ITE-operator, with the CASE-operator every arbitrary m-valued operator can be expressed. Thus, for an implementation we only have to maintain a single operator cache, what is increasing the efficiency of the implementation. Based on the generalized Boole-/Shannon decomposition w.r.t. the top variable x of the OMDDs representing the involved functions, we are able to implement the CASE-operator with the following simple recursion scheme:

$$\begin{aligned} case(f, g_0, \ldots, g_{p-1}) \quad = \quad & (x, case(f|_{x=0}, g_0|_{x=0}, \ldots, g_{p-1}|_{x=0}), \ldots \\ & \ldots, case(f|_{x=p-1}, g_0|_{x=p-1}, \ldots, g_{p-1}|_{x=p-1}). \end{aligned}$$

6.2 Mod-P Decision Diagrams

As in the case of OBDDS, OMDDs can not guarantee a concise representation of an arbitrary multiple valued function. This deficit is motivating the need of potentially more powerful data structures that can be achieved by extending the OMDD data definition. Also for OMDDs we have the possibility to introduce operator nodes that are representing a distinct multi valued function $\omega : M^p \rightarrow M$. In the case of OBDDs we have preferred to introduce $\oplus$-nodes, because the $\oplus$-operation guarantees an efficient probabilistic equivalence test. For a generalization, the $\oplus$-operation we may also be considered as being the arithmetic sum of the values represented by its successors modulo 2. Therefore, for the p-valued case, we obtain the operator $\oplus_p$, which computes the sum of all its p successors modulo p, where p must be a prime.

Definition 6.3 *Let $X = \{x_0, \ldots, x_{n-1}\}$ be a set of n p-valued variables and let $\pi : X \rightarrow \{0, 1, 2, \ldots, |X| = n-1\}$ be a bijective mapping of the variable indices. A* **Mod-p Decision Diagram** *(Mod-p-DD) P_f representing a multiple valued function $f : M^n \rightarrow M$ is a rooted, directed acyclic graph $P_f = (V, E)$ with the following properties: The node set V is containing two different types of nodes: terminal and non terminal nodes. A terminal node v_T is labeled with a constant value $c \in M$. A non terminal node v has as attributes either a variable $l(v) = x_i \in X$ (branching node), or the p-ary 'operation addition modulo p', p - prime, ($\oplus_p$-node). Each non terminal node has exactly p successors $succ_i(v) \in V, i \in M$.*

Definition 6.4 *A Mod-p-DD is called* **ordered**, *when the order in which the variables occur is consistent with π, i.e. if there is an edge leading from a branching node v_1 labeled with $l(v_1) = x_i$ to another branching node v_2 labeled with $l(v_2) = x_j$, then $\pi(x_i) < \pi(x_j)$ must hold. On each path from the root to a terminal node all variables must occur at most once.*

However, by introducing functional nodes into MDDs, we have to give up canonicity. For a fixed variable order π, a function f can be represented by several

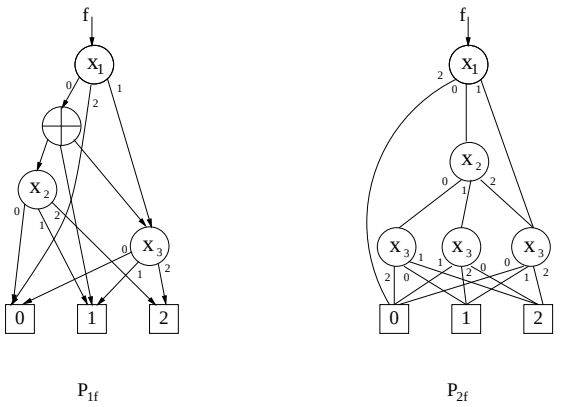


Figure 6.3: Two Different Mod-p-DDs P_{1f} and P_{2f}, both representing the same function f.

different, not isomorphic Mod-p-DDs $P_{1f}, P_{2f}, \ldots, P_{qf}$, consisting out of different subfunctions.
As an illustration, consider the following 3-variable 3-valued function $f : M^3 \rightarrow M$, $M = \{0, 1, 2\}$ defined by the following table

$x_3 \backslash x_1 x_2$	00	01	02	12	11	10	20	21	22
0	1	2	0	0	0	0	0	0	0
1	2	0	1	1	1	1	0	0	0
2	0	1	2	2	2	2	0	0	0

Two different Mod-p-DDs P_{1f}, P_{2f} representing the function f for the variable order $\pi = (1, 2, 3)$ are shown in Fig. 6.3. The functional nodes are represented by "$\oplus$". The edges leading to the three successor nodes of non terminal branching nodes are labeled with 0,1, and 2. The Mod-p-DD on the right does not contain any functional nodes, i.e. it is equivalent to the MDD of the function.
For Mod-p-DDs the set of reduction rules introduced for MDDs can be applied. For every branching node $v \in V$ the deletion rule and the merging rule are applicable. For $\oplus_p$-nodes the set of reduction rules has to be extended. In the following, consider a $\oplus_p$-node v of a Mod-p-DD P. v is connected to its p successors $v_1, \ldots, v_p$.

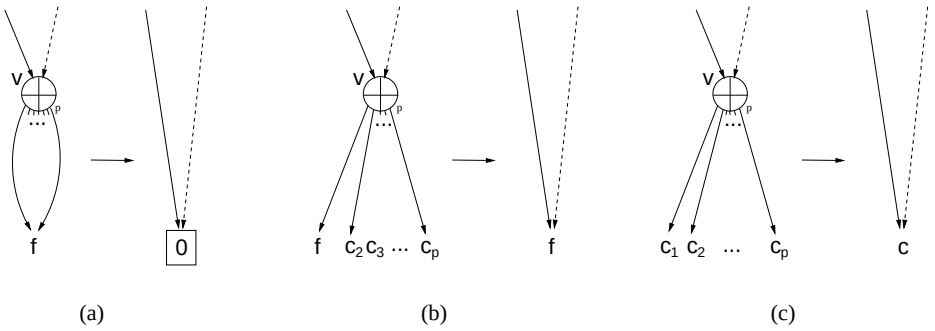
(a) If all the successors $v_1, \ldots, v_p$ of v are representing the same function and thus, $v_1 = v_2 = \ldots = v_p$, the function

$$\bigoplus_{i=1}^{p}{}_p \; f_{v_i} = \bigoplus_{i=1}^{p}{}_p \; f = p \cdot f \, (mod \; p) = 0.$$

is computed. Therefore, we can redirect all incoming edges of v to the 0-sink.

(b) Let the successors $v_1, \ldots, v_p$ of v represent the functions $f_{v_1} = f$, $f_{v_i} = c_i$, $2 \leq i \leq p$, $c_i \in M$, with the property

$$\bigoplus_{i=2}^{p}{}_p \; c_i = 0.$$



(a) (b) (c)

Figure 6.4: Three Extended Reduction Rules for Mod-p-DDs.

Then, v represents the function

$$f_v = f \oplus_p \left(\bigoplus_{i=2}^{p}{}_p \; c_i \right) = f \oplus_p 0 = f.$$

Thus, we are able to redirect all incoming edges of v to its successor v_1

(c) Let the successors $v_1, \ldots, v_p$ of v represent the functions $f_(v_i) = c_i$, $1 \leq i \leq p$, $c_i \in M$. Then v represents the function

$$\bigoplus_{i=1}^{p} {}_p c_i = c \in M$$

All incoming edges to v can be redirected to the terminal node v_{T_c} that is representing the constant function $f(v_{t_c}) = c$.

See Fig.6.4 for an illustration of the extended reduction rules for Mod-p-DDs.

Definition 6.5 *A Mod-p-DD P is called reduced, if it contains no node $v \in V$ with $succ_i(v) = succ_j(v)$ for all $i, j \in M, i \neq j$, nor does it contain distinct vertices v and v' such that the subgraphs rooted by v and v' are isomorphic. Additionally, none of the extended reduction rules for $\oplus_p$-nodes can be applied to P.*

The function $f : M^n \to M$ that is associated with the Mod-p-DD P is determined in the following way:

Definition 6.6 *A Mod-p-DD P with root node v represents a function $f : M^n \to M$ recursively as follows:*

(1) If v is a terminal node that is associated with the constant $\delta_i \in M, i \in \{0, \ldots, p-1\}$, then $f_v = \delta_i$.

(2) If v is a non terminal branching node labeled with $l(v) = x_i \in X$, then f_v is the function

$$f_v(x_1, \ldots, x_n) = \overset{0}{x_i} \; f_{succ_0(v)} + \overset{1}{x_i} \; f_{succ_1(v)} + \cdots + \overset{p-1}{x_i} \; f_{succ_{p-1}(v)},$$

where "+" and "·" denote the multiple valued operations MAX and MIN, correspondently.

(3) If v is a $\oplus_p$-node, then f_v is the function

$$f_v(x_1, \ldots, x_n) = f_{succ_0(v)} \oplus_p f_{succ_1(v)} \oplus_p \cdots \oplus_p f_{succ_{p-1}(v)},$$

where $\oplus_p$ denotes addition modulo p.

It is easy to see that MDDs are nothing else but a special case of Mod-p-DDs, namely Mod-p-DDs without $\oplus_p$-nodes in the same way as OBDDs are special $\oplus$-OBDDs. Therefore, for a given variable order π, the size of a Mod-p-DD P_f for an arbitrary multiple valued function f must be always less or equal the size of an OBDD O_f, $|P_f| \leq |O_f|$.

6.3 Probabilistic Equivalence Test for Mod-p-DDs and Finite Functions

Since Mod-p-DDs do not provide a canonical representation of multiple valued functions, testing the equivalence of two graphs becomes an essential problem. In this section we show that the equivalence of finite functions given in terms of Mod-p-DDs can be decided probabilistically in linear time by extending the probabilistic equivalence test for $\oplus$-OBDDs to the multiple valued case.
First, we introduce the concept of *multiple valued signatures* for testing the equivalence of two multiple valued functions in a probabilistic way. We define an arithmetic transformation A_m, $m \leq p$, which converts a multiple valued function $f : M^n \rightarrow M$ into a polynomial $A_m[f] : \mathbb{Z}_p^n \rightarrow \mathbb{Z}_p$ over a finite field of integers $\mathbb{Z}_p$ modulo p, for some prime p. This polynomial is used to generate a hash code for f, by evaluating the value of $A_m[f](x_1, \ldots, x_n)$ for randomly chosen input values of $x_i \in \mathbb{Z}_p$, $i \in \{1, \ldots, n\}$, because the explicit equivalence test for two polynomials would not be feasible for real applications.
Since a polynomial is unique for a given function, if two hash codes are different, then the functions are certainly not equivalent, $A_m[f_1] \neq A_m[f_2] \Rightarrow f_1 \neq f_2$. On the other hand, the hash codes of two equivalent functions are always the same, $f_1 = f_2 \Rightarrow A_m[f_1] = A_m[f_2]$ and thus, the equivalence of two arbitrary multiple valued functions can be computed with a known error probability, arising from collisions of hash codes of non equivalent functions.

6.3.1 Definition and Properties of the Generalized *A*-transform

To define the transformation A_m, we associate a *key polynomial* with each of the p^n input assignments of a multiple-valued function $f(x_1, \ldots, x_n)$. Then, all key polynomials of assignments producing the non-zero output value of f are summed up and the result is interpreted as an integer valued function $A_m[f](x_1, \ldots, x_n)$ over $\mathbb{Z}_p$.
The key polynomial for a given row of the truth table of an arbitrary multiple valued function f is a product of terms, where each term is associated with a particular input variable x_i, $i \in \{1, \ldots, n\}$. If b_i represents the value of x_i in a given row of the truth table, then the corresponding term $w(b_i, x_i)$ in the key polynomial is defined as follows:

Definition 6.7 *For any prime $m > 1 > 1$, $w : \mathbb{Z}_p \times \mathbb{Z}_p \to \mathbb{Z}_p$ is defined by*

$$w(b,x) = \sum_{i=0}^{m-1} \left(\prod_{j \in M-\{i\}} \frac{j-b}{j-i} \cdot \prod_{j \in M-\{i\}} \frac{j-x}{j-i} \right).$$

It is easy to see that $\prod_{j \in M-\{i\}} \frac{j-b}{j-i} = 1$ for $b = i$ and $\prod_{j \in M-\{i\}} \frac{j-b}{j-i} = 0$ for $b \neq i$. Therefore, parameter b acts as a selector between the terms $\prod_{j \in M-\{i\}} \frac{j-x}{j-i}$ for different values of $i \in M$, i.e.

- $w(0,x) = \prod_{j \in M-\{0\}} \frac{j-x}{j}$,
- $w(1,x) = \prod_{j \in M-\{1\}} \frac{j-x}{j-1}$,
- and so on.

On the other hand, each of the terms $\prod_{j \in M-\{i\}} \frac{j-x}{j-i}$, represents a polynomial, which computes 1 for $x = i$ and computes 0 for $x \in M - \{i\}$. Thus, the behavior of this polynomial is similar to the literal operator $\overset{i}{x}$:

$$\overset{i}{x} = \begin{cases} m-1 & \text{if } x = i, \\ 0 & \text{otherwise,} \end{cases}$$

except that for $x = i$ the literal $\overset{i}{x}$ evaluates to $m-1$, and not to 1.
Since the definition of $w(b,x)$ includes a division operation, it has to be shown that the value $w(b,x)$ always computes an integer $w(b,x) \in \mathbb{Z}_p$. As shown above, the term $\prod_{j \in M-\{i\}} \frac{j-b}{j-i}$ computes always 0 or 1, depending on b. Thus, it has to be shown that $\prod_{j \in M-\{i\}} \frac{j-x}{j-i} \in \mathbb{Z}_p$, what can be proved by the following Lemma.

Lemma 6.1 *For any $i \in M$ and any $x \in \mathbb{Z}_p$, $\prod_{j \in M-\{i\}} \frac{j-x}{j-i} \in \mathbb{Z}_p$.*

Proof: For any fixed $i \in M$ and any $x \in \mathbb{Z}_p$, $\prod_{j \in M-\{i\}} \frac{j-x}{j-i}$ has the following structure:

$$\frac{(0-x) \cdot (1-x) \cdot \ldots \cdot ((i-1)-x)}{(0-i) \cdot (1-i) \cdot \ldots \cdot ((i-1)-i)} \cdot \frac{((i+1)-x) \cdot \ldots \cdot ((m-1)-x)}{((i+1)-i) \cdot \ldots \cdot ((m-1)-i)}$$

Note that no term in the divisors can be 0 since, i is fixed and $j \neq i$. If any of the terms of the dividends is 0, the Lemma trivially holds. Thus, the case, if none of the terms in the dividends is 0 has to be considered.

By multiplying all the terms of the first fraction by -1 and reversing their order, we get:

$$\frac{(x-(i-1))\cdot\ldots\cdot(x-1)\cdot x}{1\cdot 2\cdot\ldots\cdot(i-1)\cdot i}\cdot\frac{((i+1)-x)\cdot\ldots\cdot((m-1)-x)}{1\cdot 2\cdot\ldots\cdot((m-1)-i)}$$

It is easy to see that

$$\frac{(x-(i-1))\cdot\ldots\cdot(x-1)\cdot x}{1\cdot 2\cdot\ldots\cdot(i-1)\cdot i}=\left(\begin{array}{c}x\\ i\end{array}\right)$$

and

$$\frac{((i+1)-x)\cdot\ldots\cdot((m-1)-x)}{1\cdot 2\cdot\ldots\cdot((m-1)-i)}=\left(\begin{array}{c}(m-1)-x\\ (m-1)-i\end{array}\right)$$

that are in known be integers, as long as $x, i, (m-1)-x$, and $(m-1)-i$ are integers. Since, $i \in M$ and $x \in \mathbb{Z}_p$, $x, i, (m-1)-x$ and $(m-1)-i$ are always integers and therefore, $\prod_{j\in M-\{i\}}\frac{j-x}{j-i} \in \mathbb{Z}_p$. □

The *key polynomial* W_n for an assignment $(b_1,\ldots,b_n) \in M^n$ of n variables is defined as the product of the $w(b_i, x_i)$ terms, $i \in \{1,\ldots,n\}$:

Definition 6.8 *For any $n \geq 0$ and and a prime $p > 1$, the key polynomial $W_n : Z_p^{2n} \to Z_p$ is defined by*

$$W_n(b_1,\ldots,b_n,x_1,\ldots,x_n)=\prod_{i=1}^{n} w(b_i,x_i).$$

For example, for $m = 3$:

$$W_2(0,1,x_1,x_2)=\frac{1}{2}(1-x_1)(2-x_1)x_2(2-x_2).$$

Similarly:

$$W_2(1,2,x_1,x_2)=x_1(2-x_1)(-\frac{1}{2}x_2)(1-x_2).$$

Now, we define the transformation A_m as the sum of the key polynomials W_n for all assignments $(b_1,\ldots,b_n) \in M^n$, each multiplied by the value of f for the corresponding assignment. The given definition is applicable to general functions $\mathbb{Z}_p^n \to \mathbb{Z}_p$ over the field $\mathbb{Z}_p$. Note that since $M \subset Z_p$, the multiple valued functions $f_M : M^n \to M$ are a subset of the field functions $f_{Z_p} : \mathbb{Z}_p^n \to \mathbb{Z}_p$. While a field function f_{Z_p} is as well defined for inputs $x_i \notin M$, the values that the function produces for such assignments do not participate in the definition. To distinguish between multiple valued and field functions, we use the unsubscribed letters f, g for multiple valued functions of type $M^n \to M$ and the subscribed letters f_{Z_p}, g_{Z_p} for field functions of type $\mathbb{Z}_p^n \to \mathbb{Z}_p$.

Definition 6.9 *Given a function of type* $f_{Z_p} : \mathbb{Z}_p^n \to \mathbb{Z}_p$ *and* $m > 1$, *the polynomial* $A_m[f_{Z_p}] : \mathbb{Z}_p^n \to \mathbb{Z}_p$ *is defined by*

$$A_m[f_{Z_p}](x_1, \ldots, x_n) = \sum_{\forall (b_1,\ldots,b_n) \in M^n} f_{Z_p}(b_1, \ldots, b_n) \cdot W_n(b_1, \ldots, b_n, x_1, \ldots, x_n).$$

For example, for $p = 3$ and $n = 2$, the polynomial $A_3[f_{Z_p}](x_1, x_2)$ is given by

$$\begin{aligned} A_3[f_{Z_p}](x_1, x_2) &= f(0,0) \cdot \tfrac{1}{2}(1-x_1)(2-x_1)(1-x_2)(2-x_2) + \\ &+ f(0,1) \cdot \tfrac{1}{2}(1-x_1)(2-x_1)x_2(2-x_2) + \\ &+ f(0,2) \cdot \tfrac{1}{4}(1-x_1)(2-x_1)(-x_2)(1-x_2) + \\ &+ f(1,0) \cdot x_1(2-x_1)(1-x_2)(2-x_2) + \\ &+ f(1,1) \cdot x_1(2-x_1)x_2(2-x_2) + \\ &+ f(1,2) \cdot x_1(2-x_1)(-x_2)(1-x_2) + \\ &+ f(2,0) \cdot \tfrac{1}{2}(-x_1)(1-x_1)(1-x_2)(2-x_2) + \\ &+ f(2,1) \cdot \tfrac{1}{2}(-x_1)(1-x_1)x_2(2-x_2) + \\ &+ f(2,2) \cdot \tfrac{1}{4}(-x_1)(1-x_1)(-x_2)(1-x_2). \end{aligned}$$

E.g. for the 3-valued function $f(x_1, x_2) = MIN(x_1, x_2)$, the corresponding polynomial computes to

$$\begin{aligned} A_3[f](x_1, x_2) &= x_1(2-x_1)x_2(2-x_2) + x_1(2-x_1)(-x_2)(1-x_2) + \\ &+ (-x_1)(1-x_1)x_2(2-x_2) + 2(-x_1)(1-x_1)(-x_2)(1-x_2) \\ &= \tfrac{5}{2}x_1x_2 - x_1x_1^2 - x_1^2x_2 + \tfrac{1}{2}x_1^2x_2^2. \end{aligned}$$

Note that the A-transform is defined only for assignments $(b_1, \ldots, b_n) \in M^n$. Therefore, if two field functions f_{Z_p} and g_{Z_p} have the same values for all $(b_1, \ldots, b_n) \in M^n$, then they are treated identically by the A-transform. We call these two functions to be m-equivalent, in correspondence to the A-transform previously defined for Boolean functions:

Definition 6.10 *The functions* $f_{Z_p} g_{Z_p} : \mathbb{Z}_p^n \to \mathbb{Z}_p$ *are m-equivalent if and only if* $f_{Z_p}(b_1, \ldots, b_n) = g_{Z_p}(b_1, \ldots, b_n)$ *for all assignments* $(b_1, \ldots, b_n) \in M^n$.

We write $f_{Z_p} \stackrel{m}{=} g_{Z_p}$ to denote that f_{Z_p} and g_{Z_p} are m-equivalent. If both f and g are multiple valued functions of type $M^n \to M$, then $f \stackrel{m}{=} g$ is identical to $f = g$.
By Definition 6.9, $f_{Z_p} \stackrel{m}{=} g_{Z_p}$ implies $A_m[f_{Z_p}] = A_m[f_{Z_p}]$. However, is it possible to conclude $A_m[f_{Z_p}] \neq A_m[f_{Z_p}]$ from $f_{Z_p} \stackrel{m}{\neq} g_{Z_p}$? To answer this question we have to examine the behavior of the polynomial $A_p[f]$ evaluated for some assignment $(b_1, \ldots, b_n) \in M^n$. It is easy to see that for any $b, b' \in M$, $w(b, b') = 1$, if $b = b'$, and $w(b, b') = 0$, otherwise. Therefore, for any $(b_1, \ldots, b_n), (b'_1, \ldots, b'_n) \in M^n$, $W(b_1, \ldots, b_n, b'_1, \ldots, b'_n) = 1$, if $b_i = b'_i$ for all $i \in \{1, \ldots, n\}$, and $W(b_1, \ldots, b_n, b'_1, \ldots, b'_n) = 0$, otherwise. Using these facts, the following theorem can be proven:

Theorem 6.1 *For any function* $f_{Z_p} : Z_p^n \to Z_p$, *it holds that* $A_m[f_{Z_p}] \stackrel{m}{=} f_{Z_p}$.

Proof: By Definition 6.9, for any $(b'_1, \ldots, b'_n) \in M^n$ we have:

$$A_m[f_{Z_p}](b'_1, \ldots, b'_n) = \sum_{\forall (b_1,\ldots,b_n) \in M^n} f_{Z_p}(b_1, \ldots, b_n) \cdot W_n(b_1, \ldots, b_n, b'_1, \ldots, b'_n).$$

Since, $W(b_1, \ldots, b_n, b'_1, \ldots, b'_n) = 1$ only if $b_i = b'_i$ for all $i \in \{1, \ldots, n\}$, and $W(b_1, \ldots, b_n, b'_1, \ldots, b'_n) = 0$, otherwise, this leads us to $A_m[f_{Z_p}](b'_1, \ldots, b'_n) = f(b'_1, \ldots, b'_n) \cdot 1$. Since, this equation holds for any $(b'_1, \ldots, b'_n) \in M^n$, we get $A_m[f_{Z_p}] \stackrel{m}{=} f_{Z_p}$.

□

By theorem 6.1 we are able to conclude that, though applying the A-transform to a multiple valued function f increases the domain of f from M^n to Z_p^n, the polynomial $A_m[f]$ still yields the same values as f when evaluated for an assignment $(b_1, \ldots, b_n) \in M^n$. Therefore, the polynomials for two different multiple valued functions f and g differ on all assignments $(b_1, \ldots, b_n) \in M^n$ for which $f(b_1, \ldots, b_n) \neq g(b_1, \ldots, b_n)$. Consequently, $f \neq g$ implies $A_m[f] \neq A_m[g]$.

6.3.2 Efficient Computation of the A-transform

Computing A-transforms using Definition 6.9 is only feasible for very small functions. For larger functions, we develop an alternative method, which is described in this section.

Let $f_{x_i=j}$ denote a subfunction of the $f(x_1, \ldots, x_n)$ with the variable x_i being fixed to the value j, i.e. $f_{x_i=j} \stackrel{df}{=} f(x_1, \ldots, x_{i-1}, j, x_{i+1}, \ldots, x_n)$. In the following Theorem a decomposition of $A_m[f]$ that expresses the polynomial of the function $f(x_1, \ldots, x_n)$ in terms of the polynomials of the subfunctions $f_{x_i=j}$, $i \in \{1, \ldots, n\}$, $j \in M$.

Theorem 6.2 *Every polynomial $A_m[f]$, $m > 1$, can be decomposed with respect to a variable x_i of f, $i \in \{1, \ldots, n\}$, by*

$$A_m[f] = \sum_{j=0}^{m-1} \left(\prod_{\forall k \in M - \{j\}} \frac{k - x_i}{k - j} \right) \cdot A_m[f_{x_i=j}].$$

Proof: In order to simplify the exposition and w.l.o.g. the proof is shown for

the case of $x_i = x_1$:

$$
\begin{aligned}
A_m[f] &= \sum_{\forall (b_1,\ldots,b_n)\in M^n} f(b_1,\ldots,b_n)\cdot W_n(b_1,\ldots,b_n,x_1,\ldots,x_n) && \{\text{Dfn. 6.9}\}\\
&= \sum_{j=0}^{m-1} \sum_{\forall (b_2,\ldots,b_n)\in M^{n-1}} f(j,b_2,\ldots,b_n)\cdot W_n(j,b_2,\ldots,b_n,x_1,\ldots,x_n) && \{\text{re-grouping}\}\\
&= \sum_{j=0}^{m-1} \sum_{\forall (b_2,\ldots,b_n)\in M^{n-1}} f(j,b_2,\ldots,b_n)\cdot \left(\prod_{\forall k\in M-\{j\}} \tfrac{k-x_1}{k-j}\right)\cdot W_{n-1}(b_2,\ldots,b_n,x_2,\ldots,x_n)\\
& && \{w(j,x_1) = \prod_{\forall k\in M-\{j\}} \tfrac{k-x_1}{k-j},\ \text{Dfn. 6.8}\}\\
&= \sum_{j=0}^{m-1} \left(\prod_{\forall k\in M-\{j\}} \tfrac{k-x_1}{k-j}\right)\cdot A_m[f_{x_1=j}] && \{\text{Dfn. 6.9}\}
\end{aligned}
$$

□

Next, a lemma is proven that demonstrates how the term $\prod_{\forall k\in M-\{j\}} \frac{k-x}{k-j}$, $j \in M$, can be expressed by an m-equivalent polynomial in linear form.

Lemma 6.2 *For any variable $x \in \mathbb{Z}_p$, any fixed $j \in M$, and $m > 2$ the following equation holds:*

$$
\prod_{\forall k\in M-\{j\}} \frac{k-x}{k-j} \stackrel{m}{\equiv} \begin{cases} \sum_{i=0}^{m-1} a_{i0}x^i & \text{if } j = 0,\\ a_{jj}x^j + a_{j(p-j)}x^{m-j} & \text{if } j \in M-\{0\} \text{ and } j \neq m-j,\\ a_{jj}x^j & \text{if } j \in M-\{0\} \text{ and } j = m-j. \end{cases}
$$

where $\forall i,j \in M$, $a_{ij} = \frac{D_{ij}}{D}$, with D and D_{ij} given by

$$
D = \prod_{i=1}^{m-1} i^i - \prod_{i=1}^{m-1} i^{(m-i)},
$$

$$
D_{ij} = \begin{cases} \prod_{k=1}^{m-1} k^{k\oplus_p i} - \prod_{k=1}^{m-1} k^{(m-k)\oplus_p i} & \text{if } j = 0,\\ \frac{1}{i^i}\cdot \prod_{k=1}^{m-1} k^k & \text{if } j \neq 0 \text{ and } j = i \text{ and } j \neq m-i,\\ -\frac{1}{(m-i)^i}\cdot \prod_{k=1}^{m-1} k^{m-k} & \text{if } j \neq 0 \text{ and } j = m-i \text{ and } j \neq i,\\ \frac{1}{i^i}\cdot \left(\prod_{k=1}^{m-1} k^k - \prod_{k=1}^{m-1} k^{m-k}\right) & \text{if } j \neq 0 \text{ and } j = i \text{ and } j = m-i,\\ 0 & \text{otherwise,} \end{cases} \tag{6.1}
$$

where " $\oplus_p$ " denotes addition modulo p and all other operations are regular arithmetic operations in $\mathbb{Z}_p$.

Proof: We compute the coefficients a_{ij}, $i,j \in M$, by solving the following system of m linear equations with m unknown elements:

$$\begin{cases} a_{0j}\cdot 0^0 & + a_{1j}\cdot 0^1 & + a_{2j}\cdot 0^2 & + \ldots + a_{(p-1)j}\cdot 0^{p-1} & = b_0 \\ a_{0j}\cdot 1^0 & + a_{1j}\cdot 1^1 & + a_{2j}\cdot 1^2 & + \ldots + a_{(p-1)j}\cdot 1^{p-1} & = b_1 \\ a_{0j}\cdot 2^0 & + a_{1j}\cdot 2^1 & + a_{2j}\cdot 2^2 & + \ldots + a_{(p-1)j}\cdot 2^{p-1} & = b_2 \\ \ldots & & & & \\ a_{0j}\cdot (m-1)^0 & + a_{1j}\cdot (m-1)^1 & + a_{2j}\cdot (m-1)^2 & + \ldots + a_{(m-1)j}\cdot (m-1)^{m-1} & = b_{m-1} \end{cases}$$

where $\forall i \in M$, $b_i = \prod_{\forall k \in M-\{j\}} \frac{k-i}{k-j}$. A system of linear equations like the given one above can be described by matrices as $\mathbf{X}\cdot\mathbf{a} = \mathbf{b}$, where

$$\mathbf{X} = \begin{pmatrix} 0^0 & 0^1 & 0^2 & \ldots & 0^{m-1} \\ 1^0 & 1^1 & 1^2 & \ldots & 1^{m-1} \\ 2^0 & 2^1 & 2^2 & \ldots & 2^{m-1} \\ \ldots & \ldots & \ldots & \ldots & \\ (m-1)^0 & (m-1)^1 & (m-1)^2 & \ldots & (m-1)^{m-1} \end{pmatrix},$$

$$\mathbf{a} = \begin{pmatrix} a_{0j} \\ a_{1j} \\ a_{2j} \\ \ldots \\ a_{(m-1)j} \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \ldots \\ b_{m-1} \end{pmatrix}.$$

From linear algebra it is known that a system of linear equations as the one given above always has a solution that is unique[BM77].
We compute the ith element of $\mathbf{a}$ by applying Kramer's rule, which says that, for any $i \in M$, a_{ij} is given by the formula $a_{ij} = \frac{D_{ij}}{D}$, where D is the *determinant* of $\mathbf{X}$, and D_{kj} is the determinant that is computed after the replacement of the ith column of $\mathbf{X}$ by vector $\mathbf{b}$.
Observe that the structure of matrix $\mathbf{X}$ is rather regular, namely for all $i,j \in M$, $x_{ij} = i^j$. Therefore, by applying standard rules for computing determinants [BM77], it is easy to show that D and D_{ij} are given by the equation (6.1).
Examining the structure of D_{ij}, the following properties of the elements of a_{ij} can be derived:

$$a_{ij} = \begin{cases} \frac{D_{ij}}{D} & \forall i,j: \text{ such that } j=0, \text{ or } i=j, \text{ or } i=m-j, \\ 0 & \text{otherwise.} \end{cases} \tag{6.2}$$

Thus, the only elements a_{ij} that can possibly have non-zero values, $a_{ij}! = 0$, are a_{i0} for all $i \in M$, and a_{jj} and $a_{j(m-j)}$ for all $j \in M - \{0\}$. Therefore, the term $\prod_{\forall k \in M-\{j\}} \frac{k-x}{k-j}$ can be simplified to:

$$\prod_{\forall k \in M-\{j\}} \frac{k-x}{k-j} \stackrel{m}{=} \begin{cases} \sum_{i=0}^{m-1} a_{i0}x^i & \text{if } j=0, \\ a_{jj}x^j + a_{j(m-j)}x^{m-j} & \text{if } j \in M-\{0\} \text{ and } j \neq m-j, \\ a_{jj}x^j & \text{if } j \in M-\{0\} \text{ and } j = m-j. \end{cases}$$

□

As mentioned above, the behavior of $\prod_{\forall k \in M-\{j\}} \frac{k-x}{k-j}$ is similar to the literal operator $\overset{j}{x}$, except that for $x = j$ the literal $\overset{j}{x}$ evaluates to $m-1$, and not to 1. Therefore, $\overset{j}{x} \overset{m}{=} (m-1) \cdot \prod_{\forall k \in M-\{j\}} \frac{k-x}{k-j}$, and thus, from Lemma 6.2, we can conclude that

$$\overset{j}{x} \overset{m}{=} \begin{cases} (m-1) \cdot \sum_{i=0}^{m-1} a_{0i} x^i & \text{if } j = 0, \\ (m-1) \cdot (a_{jj} x^j + a_{j(m-j)} x^{m-j}) & \text{if } j \in M - \{0\} \text{ and } j \neq m-j, \\ (m-1) \cdot a_{jj} x^j & \text{if } j \in M - \{0\} \text{ and } j = m-j. \end{cases}$$

Theorem 6.2 and Lemma 6.2 can now be used to derive another type of decomposition of $A_m[f]$ that will be used later to derive a canonical expansion for $A_m[f]$. But first, we summarize some properties of $A_m[f]$ that can be directly related to Definition 6.9.

Lemma 6.3 *For any field functions f_{Z_p} and g_{Z_p}, and any constant $c \in \mathbb{Z}_p$, it holds that*

(a) $A_m[c \cdot f_{Z_p}] = c \cdot A_m[f_{Z_p}]$.

(b) $A_m[f_{Z_p} + g_{Z_p}] = A_m[f_{Z_p}] + A_m[g_{Z_p}]$.

Theorem 6.3 *Every polynomial $A_m[f]$, $m > 1$, can be decomposed with respect to a variable x of the support of f in the following way:*
case 1: *if m is odd, then*

$$\begin{aligned} A_m[f] \;=\; & a_{00} A_m[f_{x=0}] + \\ & \sum_{j=1}^{m-1} \Big((a_{j0} A_m[f_{x=0}] + a_{jj} A_m[f_{x=j}] + a_{j(p-j)} A_m[f_{x=p-j}]) \cdot x^j \Big) \end{aligned}$$

case 2: *if m is even, then*

$$\begin{aligned} A_m[f] \;=\; & a_{00} A_m[f_{x=0}] + \\ & \sum_{\substack{j=1 \\ j \neq m/2}}^{m-1} \Big((a_{j0} A_m[f_{x=0}] + a_{jj} A_m[f_{x=j}] + a_{j(m-j)} A_m[f_{x=m-j}]) \cdot x^j + \\ & + \; (a_{\frac{m}{2}0} A_m[f_{x=0}] + a_{\frac{m}{2}\frac{m}{2}} A_m[f_{x=m/2}]) \cdot x^{m/2} \Big) \end{aligned}$$

where $\forall i, j \in M$, $a_{ij} = \frac{D_{ij}}{D}$, and D and D_{ij} given by (6.1).

Proof:

$$
\begin{aligned}
A_m[f] &= \sum_{j=0}^{m-1} \left(\prod_{\forall k \in M-\{j\}} \frac{k-x_i}{k-j} \right) \cdot A_m[f_{x_i=k}] && \{\text{Theorem 6.2}\} \\
&= \sum_{i=0}^{m-1} a_{i0} x^i \cdot A_m[f_{x=0}] + \\
&\quad \sum_{j=1}^{p-1} (a_{jj} x^j + a_{j(p-j)} x^{p-j}) \cdot A_m[f_{x=j}] && \{\text{Lemma 6.2}\} \\
&= a_{00} A_m[f_{x=0}] + \\
&\quad \sum_{j=1}^{m-1} (a_{j0} A_m[f_{x=0}] + a_{jj} A_m[f_{x=j}] + a_{j(m-j)} A_m[f_{x=m-j}]) \cdot x^j \\
& && \{\text{reordering}\}
\end{aligned}
$$

□

Let F be the vector of coefficients of the truth table of the function f and A^n be a transformation matrix, defined as follows:

Definition 6.11 *The $m^n \times m^n$ matrix A^n is defined defined inductively by:*

(1) $$A^1 \stackrel{df}{=} \begin{bmatrix} a_{00} & 0 & 0 & \dots & 0 & 0 \\ a_{10} & a_{11} & 0 & \dots & 0 & a_{1(m-1)} \\ a_{20} & 0 & a_{22} & \dots & a_{2(m-2)} & 0 \\ \dots & \dots & \dots & \dots & & \\ a_{(m-2)0} & 0 & a_{(m-2)2} & \dots & a_{(m-2)(m-2)} & 0 \\ a_{(m-1)0} & a_{(m-1)1} & 0 & \dots & 0 & a_{(m-1)(m-1)} \end{bmatrix}$$

where $\forall i, j \in M$, the coefficients a_{ij} are given by (6.2).

(2) $A^n \stackrel{df}{=} A^1 \otimes A^{n-1}$

where " $\otimes$ " denotes the Kronecker product of two matrices [BM77].

Clearly, if Theorem 6.3 is successively applied to the polynomials $A_m[f_{x_i=k}]$ of subfunctions $f_{x_i=k}$ for all the remaining variables, an expression can be derived, in which $A_m[f]$ is expanded in all the variables of f:

Theorem 6.4 *Every polynomial $A_m[f]$, $m > 2$, of an n-variable p-valued function f can be expressed in the following canonical form*

$$A_m[f] = \sum_{j=0}^{m^n-1} c_j \cdot x_1^{i_1} \cdot x_2^{i_2} \cdot \ldots \cdot x_n^{i_n},$$

where $(i_1 i_2 \dots i_n)$ is the m-ary expansion of i, with i_1 being the least significant digit, and the coefficients c_i are given by the vector $C \stackrel{df}{=} [c_0 c_1 \dots c_{p^n-1}]$ computed as $C = A^n \cdot F$.

Proof: By induction on n. We show the proof only for the case of m being odd. For m being even p the proof is similar.

(1) Let $n = 1$. According to Theorem 6.3, any polynomial $A_m[f]$ of a function $f(x)$ of one variable x can be decomposed with respect to x as:

$$\begin{aligned} A_m[f] &= a_{00} \cdot A_m[f_{x=0}] + \\ &\sum_{j=1}^{m-1} (a_{j0} \cdot A_m[f_{x=0}] + a_{jj} \cdot A_m[f_{x=j}] + a_{j(m-j)} \cdot A[f_{x=m-j}]) \cdot x^j, \end{aligned}$$

where $f_{x=k} = f(k)$. According to Lemma 6.3, $A_m[c] = c$ for any constant $c \in \mathbb{Z}_p$. Thus, the expression above can also be written as:

$$A_m[f] = a_0 0 \cdot f(0) + \sum_{j=1}^{m-1} (a_{j0} \cdot f(0) + a_{jj} \cdot f(j) + a_{j(p-j)} \cdot f(m-j)) \cdot x^j$$

which can be re-written as

$$A_m[f] = \sum_{i=0}^{m-1} c_i x^i,$$

where $c_0 = a_{00} \cdot f(0)$ and $c_j = a_{j0} \cdot f(0) + a_{jj} \cdot f(j) + a_{j(m-j)} \cdot f(m-j)$, for all $j \in M - \{0\}$. By examining the structure of the matrix A^1, one can conclude that $C = A^1 \cdot F$.

(2) Hypothesis: Assume the result for n. According to Theorem 6.3, any $A_m[f]$ of a function f of $n+1$ variables can be decomposed with respect to x_{n+1} in the following way:

$$\begin{aligned} A_m[f] &= a_{00} A_m[f_{x_{n+1}=0}] + \\ &\sum_{j=1}^{m-1} (a_{j0} A_m[f_{x=0}] + a_{jj} A_m[f_{x=j}] + a_{j(m-j)} A_m[f_{x=m-j}]) \cdot x_{n+1}^j. \end{aligned}$$

By the induction hypothesis, we can express each A_m of the subfunctions of n variables in the canonical form. We use the notation c_i^k to denote the ith coefficient of the canonical form of the subfunction $f_{x_{n+1}=k}$ and F_k for the truth table vector of $f_{x_{n+1}=k}$. To simplify the exposition we also use the abbreviation X to replace the term $x_1^{i_1} \cdot x_2^{i_2} \cdot \ldots \cdot x_n^{i_n}$. Thus, each $A_m[f_{x_{n+1}=k}]$, $k \in M$, is replaced by $\sum_{i=0}^{m-1} c_i^k \cdot X$, with c_i^k given by $C_k = A^n \cdot F_k$. Then we get:

$$\begin{aligned} A_m[f] &= a_{00} \cdot \sum_{i=0}^{m^n-1} c_i^0 \cdot X + \\ &\sum_{j=1}^{m-1} \left(a_{j0} \cdot \sum_{i=0}^{m^n-1} c_i^0 \cdot X + a_{jj} \cdot \sum_{i=0}^{m^n-1} c_i^j \cdot X + a_{j(m-j)} \cdot \sum_{i=0}^{m^n-1} c_i^{m-j} \cdot X \right) \cdot x_{n+1}^j. \end{aligned}$$

Since, " $\cdot$ " is distributive over " $+$ ", we can reorder the above equation as

$$\begin{aligned} A_m[f] &= \left(\sum_{i=0}^{m^n-1} a_{00} \cdot c_i^0 \cdot X \right) \cdot x_{n+1}^0 + \\ & \sum_{j=1}^{m-1} \left(\sum_{i=0}^{m^n-1} (a_{j0} \cdot c_i^0 + a_{jj} \cdot c_i^j + a_{j(m-j)} c_i^{m-j}) \cdot X \right) \cdot x_{n+1}^j, \end{aligned}$$

which can be rewritten as

$$A_m[f] = \sum_{j=0}^{m^n-1} c_j \cdot x_1^{i_1} \cdot x_2^{i_2} \cdot \ldots \cdot x_n^{i_n},$$

where $c_i = a_{00} \cdot c_i^0$, for $0 \leq i \leq m-1$, and $c_i = a_{j0} \cdot c_i^0 + a_{jj} \cdot c_i^j + a_{j(m-j)} c_i^{m-j}$, for $j \cdot m \leq i \leq j \cdot m + m - 1$, for all $j \in M - \{0\}$. Since, the coefficients c_i^j are given by $C_j = A^n \cdot F_j$, this is equivalent to $C = (A^1 \otimes A^n) \cdot F = A^{n+1} \cdot F$.

□

For example, for $m = 3$ and $n = 2$, the matrix A^2 is constructed as follows:

$$A^1 = \begin{bmatrix} 1 & 0 & 0 \\ -3/2 & 2 & -1/2 \\ 1/2 & -1 & 1/2 \end{bmatrix}$$

and

$$A^2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3/2 & 2 & -1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & -1 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3/2 & 0 & 0 & 2 & 0 & 0 & -1/2 & 0 & 0 \\ 9/4 & -3 & 3/4 & -3 & 4 & -1 & 3/4 & -1 & 1/4 \\ -3/4 & 3/2 & -3/4 & 1 & -2 & 1 & -1/4 & 1/2 & -1/4 \\ 1/2 & 0 & 0 & -1 & 0 & 0 & 1/2 & 0 & 0 \\ -3/4 & 1 & -1/4 & 3/2 & -2 & 1/2 & -3/4 & 1 & -1/4 \\ 1/4 & -1/2 & 1/4 & -1/2 & 1 & -1/2 & 1/4 & -1/2 & 1/4 \end{bmatrix}$$

Thus, $A_3[f]$ for $f = MIN(x_1, x_2)$ can be computed as $C = A^2 \cdot F =$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3/2 & 2 & -1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & -1 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3/2 & 0 & 0 & 2 & 0 & 0 & -1/2 & 0 & 0 \\ 9/4 & -3 & 3/4 & -3 & 4 & -1 & 3/4 & -1 & 1/4 \\ -3/4 & 3/2 & -3/4 & 1 & -2 & 1 & -1/4 & 1/2 & -1/4 \\ 1/2 & 0 & 0 & -1 & 0 & 0 & 1/2 & 0 & 0 \\ -3/4 & 1 & -1/4 & 3/2 & -2 & 1/2 & -3/4 & 1 & -1/4 \\ 1/4 & -1/2 & 1/4 & -1/2 & 1 & -1/2 & 1/4 & -1/2 & 1/4 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 5/2 \\ -1 \\ 0 \\ -1 \\ 1/2 \end{bmatrix}.$$

Finally, $A_3[f](x_1, x_2) = \frac{5}{2} x_1 x_2 - x_1 x_1^2 - x_1^2 x_2 + \frac{1}{2} x_1^2 x_2^2$.

6.3.3 Probabilistic Equivalence Test for Mod-p-DDs

In the previous section we have shown how to compute the A-transform $A_m[f]$ for an arbitrary multi valued function $f : M^n \to M$. $A_m[f]$ is unique for the given function f and thus, can be used to determine the equivalence of two functions f_1 and f_2. To simplify the task of deciding the equivalence of f_1 and f_2, the equivalence of the two polynomials $A[f_1]$ and $A[f_2]$ can be decided probabilistically by testing the equivalence for randomly chosen instances. Thus, we are computing *signatures* (hash codes) for identifying multiple valued functions with a certain probability of error.
These signatures can also be used for testing the equivalence of Mod-p-DDs probabilistically, as in the case of $\oplus$-OBDDs. For computing signatures for Mod-p-DDs, we associate a polynomial $p_{f_v} = A_m[f_v]$ with every node v of a Mod-p-DD, according to the function f_v that the node v is computing.
Let $GF(p^k)$ be a Galois Field with p^k elements of characteristic p, p - prime, $k > 0$.

Definition 6.12 *Let G be a Mod-p-DD representing a multiple valued function $f : M^n \to M$. With each node $v \in G$ we associate the* polynomial $p_v : (GF((p^k))^n \to GF(p^k)$ *defined in the following way:*

1. $p_v = \delta_i$, *if v is a terminal node representing the value $\delta_i \in M$,*
2. $p_v = \sum_{j=0}^{p-1} \left(\prod_{\forall r \in M - \{j\}} \frac{r - x_i}{r - j} \right) \cdot p_{child_j(v)}$, *if v is a non terminal branching node with $index(v) = i$,*
3. $p_v = \sum_{j=0}^{p-1} p_{child_j(v)}$, *if v is a $\oplus_p$-node,*

where the operations "$+$", "$-$" and "$\cdot$" are computed in the field $GF(p^k)$.

The *polynomial* identifying the Mod-p-DD G, p_G, is the polynomial associated with the root node of G. Since p_G is determined by the function f that is represented by G, p_G remains unchanged for different Mod-p-DDs representations of the same function.
Following Definition 6.12, p_G can be computed with $p \cdot |G|$ many additions, at most $2p^2 \cdot |G|$ many subtractions and at most $2p^2 \cdot |G|$ multiplications. Using $GF(p^k)$ of characteristic p, simplifies the polynomial for addition modulo p operation to $p_{f_1 \oplus f_2} = p_{f_1} + p_{f_2}$. If we consider the elements of $GF(p^k)$ as p-ary vectors of length k, then field addition can be computed in constant time by bitwise addition modulo p. Multiplication and subtraction of two p-ary vectors of length k can be computed in time $\lceil \log p \rceil k$. Therefore, p_G can be determined in at most $p \cdot |G| + 2p^2 \cdot |G| \cdot \lceil \log p \rceil k + 2p^2 \cdot |G| \cdot \lceil \log p \rceil k$ time. Since, p and k are constants, the complexity of computing p_G is bounded by $O(|G|)$.
During Mod-p-DD synthesis the computation of p_G is performed bottom-up and thus, the complexity of computing the polynomial for a new node takes only constant time, because the polynomials for all its successor nodes have already been computed.

Input: Mod-p-DDs P_f, P_g.
Output: If $f = g$ the algorithm answers *yes*, otherwise it returns *no* with probability greater than $\frac{1}{p}$.

```
equivalence(P_f, P_g) {
   chose independently and uniformly at random a_1,...,a_n ∈ GF(p^k);
   compute p_f(a_1,...,a_n) and p_g(a_1,...,a_n);
   if ( p_f(a_1,...,a_n)==p_g(a_1,...,a_n) ) {
     return(yes);
   } else {
     return(no);
   }
}
```

Figure 6.5: Algorithm for a Probabilistic Equivalence Test for Mod-p-DDs.

For computing the signatures of the nodes v of a Mod-p-DD G, for every variable $x_i \in \{x_1, \ldots, x_n\}$ of the multiple valued function $f : M^n \to M$ represented by G, we chose the elements $A_m[x_i] = p_{x_i} \in GF(p^k)$, $i = (1, \ldots, n)$ independently and uniformly at random. With these values p_{x_i} the hash codes $p_v(p_{x_1}, \ldots, p_{x_n})$ identifying a unique node v of the Mod-p-DD G can be computed efficiently.
Now, all prerequisites are done for defining an algorithm for the probabilistic equivalence test for Mod-p-DDs.
Let P_f and P_g be two Mod-p-DDs representing the multiple valued functions $f, g : M^n \to M$. Let $k \in \mathbb{N}$, such that $|GF(p^k)| > p \cdot n$. Assume that $a_1, \ldots, a_n \in GF(p^k)$ are generated independently and uniformly at random.

Theorem 6.5 *For the Boolean signatures p_f and p_g that are computed for the Mod-p-DDs P_f and P_g, it holds that*

(1) $p_f(a_1, \ldots, a_n) = p_g(a_1, \ldots, a_n)$, *if* $f = g$, *and*

(2) $Prob(p_f(a_1, \ldots, a_n) = p_g(a_1, \ldots, a_n)) < \frac{1}{p}$, *if* $f \neq g$.

Proof: (1) follows directly from the definition of the A-transform $A_m[f]$ for a multiple valued function f.
(2) Following the proof given in theorem 4.13 the error bound can be estimated with $Prob(p_f(a_1, \ldots, a_n) = p_g(a_1, \ldots, a_n)) < \frac{1}{p}$. □

See Fig 6.5 for the algorithm in pseudocode.
The estimation of the probability of degeneracy for Mod-p-DDs follows the same estimation for $\oplus$-OBDDs in Theorem 4.16.
By using s parallel signatures the probability of degeneracy in a Mod-p-DD G representing a multiple valued function $f : M^n \to M$ between any two nodes is at most $\frac{|P|^2 \cdot n^s}{2 \cdot |GF(2^m)|^s}$.

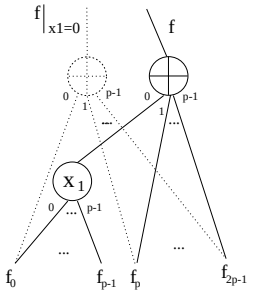


Figure 6.6: Cofactor Creation $f|_{x_1=0}$ for Mod-p-DDs.

6.4 Synthesis of Mod-P-DDs

The key operation for the construction of Mod-p-DDs is the Mod-p-DD synthesis procedure. Applying an arbitrary multiple valued operation to two Mod-p-DDs requires an extension of the CASE algorithm that has already been introduced for MDD synthesis, resulting in the so called CASE-$\oplus$ algorithm. The CASE-$\oplus$-algorithm can be implemented by recursion with the cofactors of the functions under consideration w.r.t. the top variables of the involved Mod-P-DDs. Again, as in the case of $\oplus$-OBDDs, the creation of cofactors has to be adapted for Mod-P-DDs.

6.4.1 Cofactor Creation for Mod-P-DDs

The adaption of the CASE-algorithm to the CASE-$\oplus$ algorithm necessitates the creation of Mod-p-DDs for the cofactors $f|_{x_i=j}, j \in M$ of a multiple-valued function f associated with a Mod-p-DD.
There, for Mod-p-DDs we have to distinguish two different cases:
The top node v_T of a Mod-p-DD P can either be a branching node labeled with a variable $x_i \in X_n$, or it can be a $\oplus_p$-node, connected to branching nodes, where at least one of its successors is labeled with x_i. In the first case, for creating the cofactor $f|_{x_i=j}, j \in M$, simply the successor v_{T_j} of v_T that is connected to the j- edge, has to be returned. In the other case, an $\oplus_p$-node v'_T has to be returned that is connected to the cofactors $f|_{x_i=j}$ of its successors $v_{T_1}, \ldots, v_{T_p}$. If the $\oplus_p$-node v'_T does not already exist, it has to be newly created (see Figure 6.6).

6.4.2 Extended CASE-$\oplus$ Algorithm for Mod-P-DDs

As the regular CASE algorithm, the CASE-$\oplus$-algorithm computes the result of the application of a p-ary operator $\otimes_p$ to p operands $f_1, \ldots, f_p$, $h = \otimes_p(f_1, \ldots, f_p)$, all given in terms of Mod-p-DDs. The CASE-$\oplus$ algorithm itself requires $(p+1)$ input parameters that determine the p-ary function that should be represented. Therefore, in a preprocessing step an algorithm called APPLY-$\oplus$ transforms the p operands $f_1, \ldots, f_p$ into the $(p+1)$-tuple $(f, g_0, \ldots, g_{p-1})$ that serves as input parameter for the CASE-$\oplus$-algorithm. As a special case $\otimes_p = \oplus_p$,

$h = \oplus_p(f_1, \ldots, f_p)$ has to be considered. There, in the preprocessing step, we can directly create a $\oplus_p$-node and connect it with the p Mod-p-DDs representing $f_1, \ldots, f_p$.
The extension of the CASE-algorithm from MDD synthesis to Mod-p-DDs differs naturally in the treatment of $\oplus_p$-nodes in the Mod-p-DD. There, the CASE-$\oplus$-algorithm follows closely the ITE-$\oplus$-algorithm for $\oplus$-OBDDs in the binary case. The input parameters of the CASE-$\oplus$ operator are, in general, multiple-valued functions given in the terms of Mod-p-DDs. The task is to generate the resultant function h =CASE-$\oplus(f, g_0, g_1, \ldots, g_{p-1})$ recursively.
If f represents only the variable x, $f(x) = x$, then the result $h = CASE - \oplus(x, g_1, \ldots, g_p)$ returned by CASE-$\oplus$ corresponds to a multiple valued branching node labeled with the variable x and connected to the Mod-p-DDs of $g_0, g_1, \ldots, g_{p-1}$ as successors:

$$\text{CASE-}\oplus\ (x, g_0, g_1, \ldots, g_{p-1}) = (x, g_0, g_1, \ldots, g_{p-1}).$$

Moreover, if $g_1, \ldots, g_p$ happen to be the constant functions $g_1 = 0, g_2 = 1, \ldots, g_p = p - 1$, it holds that

$$\text{CASE-}\oplus\ (f, 0, 1, \ldots, p-1) = f.$$

The above equations form the terminal cases for the recursive CASE-$\oplus$ algorithm.
If f is a complex function, then CASE-$\oplus$ of the cofactors of f w.r.t. the top variable x is called recursively, and the results are composed according to the generalized Boole/Shannon decomposition.
To speed up the performance of the CASE-$\oplus$ operation, we use a *computed table*, which is organized as a hash based cache, to store and recall the already computed results. Before a new node is created, we always refer to a *unique table* that is organized as a hash table, to prevent the creation of already allocated nodes. In both, *computed table* and *unique table*, every reference is made by identifying the underlying Mod-p-DDs by their signatures. A node v that is labeled with the variable x_i is represented by an $(p+1)$-tuple $(x_i, v_0, v_1, \ldots, v_{p-1})$, with v_j, $1 \leq j \leq p-1$ being the node connected to $child_j(v)$. To avoid redundant entries in the *computed table* we transform the $(p+1)$-tuple to a standard form by reordering it in the same way as for the binary case.
The pseudo-code of the CASE-$\oplus$ algorithm is shown in Figure 6.7. First, a preprocessing step takes care of the transformation of the input parameters are into a corresponding $(p+1)$-tuple or, for the case of performing the computation of $\oplus_p(f_1, \ldots, f_p)$ is taken care of. Then, the CASE-$\oplus$ algorithm is called and first, possible terminal cases are considered. If the resulting function has already been computed and stored in the unique table, then it will be directly returned. In the following step, the cofactors $h|_{x=j}$, $0 \leq j \leq p-1$ of the function h are computed by calling CASE-$\oplus$ recursively with the cofactors $f|_{x=j}, g_0|_{x=j}, g_1|_{x=j}, \ldots, g_{p-1}|_{x=j}$ as its arguments. The resulting Mod-p-DDs of these recursive calls are composed using the generalized Boole/Shannon decomposition as $(x, h_0|_{x=0}, h_1|_{x=1}, \ldots, h_{p-1}|_{x=p-1})$ to a new Mod-p-DD representing the required function.

Input: Mod-p-DDs $P_{f_1}, \ldots, P_{f_p}$, and an operator $\otimes_p$
Output:Mod-p-DD P_{res} representing the function $h = \otimes_p(f_1, \ldots, f_p)$.

```
CASE-⊕(f, g_0, g_1, ..., g_{p-1}) {
  transform_to_standard_tuple(f, g_0, g_1, ..., g_{p-1});
  if (terminal_case(f, g_0, g_1, ..., g_{p-1}, res)) {
    return(res);
  }
  reorder_tuple_acc_to_variable_order(f, g_0, g_1, ..., g_{p-1});
  if (in_computed_table(f, g_0, g_1, ..., g_{p-1}, res)) {
    return(res);
  }
  for (j = 0; j < (p - 1); j++) {
    h|_{x=j}=CASE-⊕(f|_{x=j}, g_0|_{x=j}, ..., g_{p-1}|_{x=j});
  }
  if (signature(h|_{x=0})==signature(h|_{x=1})==...==signature(h|_{x=p-1})) {
    res = h|_{x=0};
  } else {
    res=create_node(x, h_{x=0}, ..., h_{x=p-1});
  }
  insert_in_computed_table(f, g_0, g_1, ..., g_{p-1}, res);
  find_or_add_in_unique_table(res);
  return(res);
}


APPLY-⊕_p(P_{f_1}, ..., P_{f_p}, ⊗_p) {
  if (⊗_p == ⊕_p) {
    P_res=create_node(⊕_p, P_{f_1}, ..., P_{f_p});
    find or add in unique table(P_res);
  } else {
    transform (P_{f_1}, ..., P_{f_p}) into CASE tuple (P_f, P_{g_0}, ..., P_{g_{p-1}});
    P_res=CASE-⊕(P_f, P_{g_0}, ..., P_{g_{p-1}});
  }
  return(P_res);
}
```

Figure 6.7: CASE-$\oplus$ and APPLY-$\oplus_p$ Algorithm for Mod-p-DD Synthesis.

For a possible implementation of a Mod-p-DD package two possible way can be considered:

(1) Implementation of a Mod-p-DD package from the scratch, employing only multiple valued operations, or

(2) Implementation of Mod-p-DDs via a suitable binary encoding, employing an already existing package for binary decision diagrams.

While possibility (1) requires a new implementation from the scratch, using an already existing package as proposed in (2) might seem much more convenient, but it also causes some serious problems (see section *Possible Future Work* in the following chapter).

Chapter 7

Conclusions

7.1 Key Results

In this thesis $\oplus$-OBDDs have been presented, a data structure for the representation of Boolean functions and applicable for formal verification tasks. The necessity of extending the concept of OBDDs has been motivated, which led to the introduction of additional operator nodes into the OBDD data structure. $\oplus$ and $\equiv$ (*exclusive or* and *equivalence*) have qualified to be the most suitable operators for this task, because the required algorithmic properties of the resulting data structure can be maintained. It has been shown that $\oplus$-OBDDs are a more powerful data structure for the representation of Boolean functions compared to OBDD, FBDDs, ESOPs, or OFDDs by simulating these data structures efficiently with $\oplus$-OBDDs and by showing the existence of exponential gaps in the computational power between these data structures and $\oplus$-OBDDs.

$\oplus$-OBDDs are not a canonical data structure and a deterministic equivalence test for $\oplus$-OBDDs is not efficient enough to be employed in a practical working environment. A fast probabilistic equivalence test for $\oplus$-OBDDs with sufficient reliability has been presented and was applied in a symbolic simulation environment based on $\oplus$-OBDDs. For working with $\oplus$-OBDDs all necessary manipulation algorithms, as there are the *creation of cofactors* and the *synthesis* of $\oplus$-OBDDs have been shown in detail and their efficiency was proven by experimental results in symbolic simulation of standard benchmark sets.

For improving the efficiency of $\oplus$-OBDDs it has been shown that $\oplus$-OBDD size is dependent on $\oplus$-node frequency, $\oplus$-placement, and also on the chosen variable order. Methods for including additional $\oplus$-nodes into the $\oplus$-OBDD data structure have been presented and a heuristic has been proposed for suitable $\oplus$-node placement.

For further optimization, it has been shown, how to move $\oplus$-OBDDs efficiently within the given data structure. For this purpose, another heuristic maintaining the given variable order, while changing the position of already existing $\oplus$-nodes has been presented and confirmed by experiments.

Implementational details that are crucial for efficient dynamic reordering of $\oplus$-OBDDs have been discussed and it has been shown, how to perform dynamic reordering of $\oplus$-OBDD variables, while simultaneously taking care of existing

$\oplus$-nodes within the given data structure. The power of dynamic reordering heuristics for $\oplus$-OBDDs also has been confirmed by experimental results.
Finally, the concept of $\oplus$-OBDDs has been extended to an arbitrary finite domain and Mod-p-DDs have been presented. Because Mod-p-DDs are also not a canonical data structure, for testing the equivalence of Mod-p-DDs the fast probabilistic equivalence test for $\oplus$-OBDDs had to be extended. Additionally, a method for the efficient computation of a Boolean signature for Mod-p-DDs, as for arbitrary finite functions has been introduced. Based on this probabilistic equivalence test, all necessary manipulation algorithms for Mod-p-DDs have been developed and discussed.

7.2 Possible Future Work

For a further improvement of the efficiency of the $\oplus$-OBDD data structure, research on heuristics for $\oplus$-node placement and variable reordering could be emphasized. More sophisticated heuristics, driven by functional requirements of the function to be represented should be investigated.
For an exact minimization, the algorithm proposed by Waack for POBDDs can be adapted to $\oplus$-OBDDs. Starting in a bottom up approach all functions could be represented by a linear combination of functions that are representing basis vectors of the underlying vector space.
But, the runtime complexity of this method as being dependent on the complexity of the Gaussian elimination method for computing the basis vectors and thus, being at most cubic in the number of nodes prevents its application in a practical working environment in advance.
For $\oplus$-OBDDs also other applications besides symbolic simulation of combinatorial circuits are possible. For the formal verification of sequential designs the exploration of their state spaces is an important and mandatory task. There, the symbolic representation of the state space with its characteristic function and the transition relation for the stepwise computation of the state space could be performed with $\oplus$-OBDDs. Because of their improved power of representation, also state space exploration could possibly benefit of the usage of $\oplus$-OBDDs. Also their utilization in symbolic model checking is possible. Almost all tasks that can be performed with regular OBDDs can be adapted for the usage of $\oplus$-OBDDs.
On the other hand, we have described all necessary manipulation algorithms for Mod-p-DDs. The next step would be an efficient implementation of this data structure to be utilized in symbolic simulation of multiple valued logic designs. For an implementation of Mod-p-DDs, there are two possible alternative ways: First, we could develop a new Mod-p-DD package from the scratch, which means a direct implementation of the multiple valued data structure. But, secondly, we could also build a Mod-p-DD package on top of the already existing $\oplus$-OBDD package: For representing a multiple valued function $f : M^n \rightarrow M$ with $\oplus$-OBDDs, we encode f into a Boolean function f_B of type

$$f_B : \{0,1\}^{\lceil \log p \rceil \cdot n} \rightarrow \{0,1\}^{\lceil \log p \rceil}.$$

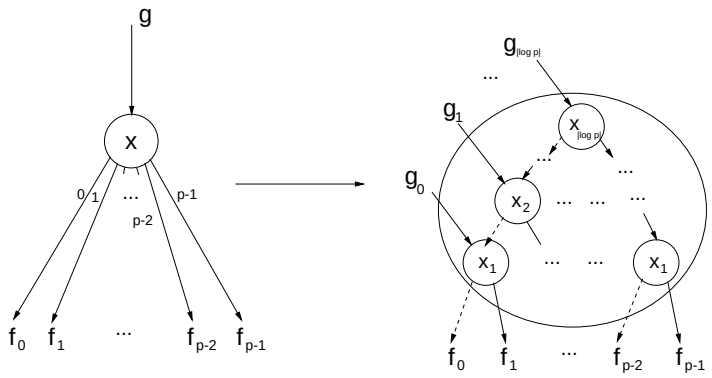


Figure 7.1: Binary Encoding of Multiple Valued Variables.

Thus, a p valued variable x_i can be simulated by $\lceil \log p \rceil$ binary variables $x_{i_1}, \ldots, x_{i_{\lceil \log p \rceil}}$ (See Fig. 7.1 for an illustrating example). This encoding has already been successfully employed for MDDs. This way of implementation does also benefit from the already implemented variable reordering techniques for the binary data structure. Reordering of multiple valued variables x_i can be simulated by group wise reordering of binary variables representing an encoding of x_i, i.e. keeping the binary variables $x_{i_1}, \ldots, x_{i_{\lceil \log p \rceil}}$ always together and performing reordering only either by relocating the whole group of binary variables or by reordering members inside of the group only.
A possible obstacle is the efficient encoding of $\oplus_p$ nodes by binary $\oplus$-nodes. Even for the case, if an efficient encoding is found, further improvement by changing the position of $\oplus$-nodes will distract the group of $\oplus$-nodes representing a $\oplus_p$ node and thus, preventing the transformation back. Thus, a binary encoding would only transform a problem given in terms of multiple valued logic to binary logic that will be represented with a binary data structure. Benefits arising form the utilization of a native multiple valued data structure as Mod-p-DDs could not be further exploited.
For taking the full advantage of Mod-p-DDs it is necessary to undergo an implementation from the scratch. There, we can get rid of all required encoding and decoding steps, and problem specifications given directly in terms of multiple valued logic can be represented and processed.

Appendix A

Experimental Results

circuit	OBDD size	⊕-OBDD size			
		ITE (⊕)	nDE (⊕)	pDE (⊕)	stan(⊕)
sbc	3715	3715	4598 (2781)	5105 (3074)	3715
s967	1732	1755	1073 (676)	1208 (738)	1755
s820	2651	2651	552 (328)	718 (418)	2651
s713	1352	1352	3554 (2114)	3122 (1626)	1352
s641	1352	1352	3550 (2116)	3089 (1609)	1352
s635	656	656	1746 (1118)	668 (44)	656
s526	232	232	371 (190)	342 (163)	232
s510	19076	19076	636 (433)	739 (503)	19076
s499	336	336	640 (123)	798 (261)	336
s444	226	226	390 (199)	399 (213)	226
s420	262227	262227	732 (431)	471 (201)	262227
s386	281	281	295 (150)	298 (140)	281
s3384	748809	748809	1142383 (781005)	1139458 (762028)	748809
s3271	3365	3365	6437 (4064)	6511 (4038)	3365
s208	1033	1033	186 (105)	141 (67)	1033
s1512	18896	18896	10941 (7295)	7148 (3499)	18896
s1494	1016	1016	1378 (979)	1349 (906)	1016
s1488	1016	1016	1316 (928)	1357 (909)	1016
s1423	98454	98454	134008 (87748)	113035 (68101)	98454
s1269	48177	48177	39922 (27974)	33418 (23231)	48177
s1196	2295	2295	3844 (2374)	3175 (1826)	2295
8085	117841	117841	107915 (70724)	85059 (52984)	117841
bigkey	6170	6173 (3)	79770 (4720)	7633 (4494)	6181 (4)
dsip	13921	13921	9675 (5629)	12363 (7867)	13921
mm4a	675	675	1439 (858)	1407 (866)	675
mm9b	848081	-	658964 (451322)	61233 (43358)	-
mm9a	735768	-	533707 (345779)	50926 (35409)	-
mult16a	360442	262188 (31)	655125 (392968)	655077 (392920)	262188 (31)
Σ	**3.299.795**	n.a	**3.405.147**	**2.196.247**	n.a.
	100%		103.2%	66.6%	

Table A.1: Comparing OBDD and ⊕-OBDD **Size** for Fixed Variable Order - Sequential Circuits.

circuit	OBDD size	⊕-OBDD size			
		ITE (⊕)	nDE (⊕)	pDE (⊕)	stan(⊕)
x3	2760	2760	2369 (1255)	2230 (1096)	2760
x1	1297	1297	2948 (1611)	2978 (1673)	1297
vg2	1044	1044	2071 (1256)	2195 (1294)	1044
vda	4345	4345	1954 (1399)	1895 (1276)	4345
too_large	7096	7096	14507 (8748)	21858 (13963)	7096
term1	580	580	1161 (726)	1059 (628)	580
pair	67685	67929 (284)	108998 (55910)	114057 (60222)	185223 (293)
my_adder	327677	262188 (30)	589831 (327674)	524297 (262139)	262188 (30)
mux	131071	131071	217 (184)	217 (173)	131071
k2	28336	28336	5986 (3955)	7479 (4920)	28336
i9	2278	2278	8754 (5132)	10258 (6509)	2278
i8	4366	4366	14750 (9762)	15307 (9896)	4366
i7	505	505	1000 (496)	889 (385)	505
i5	312	312	763 (391)	636 (324)	312
i4	421	421	1095 (674)	845 (388)	421
i2	335	335	317 (47)	851 (581)	335
frg2	6471	6472 (1)	5031 (2655)	7246 (4451)	6461
frg1	204	204	383 (172)	334 (132)	204
example2	469	469 (2)	644 (306)	485 (146)	453
count	234	234 (7)	294 (92)	308 (177)	226
cm150a	131071	131071	220 (187)	207 (174)	131071
booth8x8	6386	5959 (109)	14456 (9653)	14642 (9591)	-
baugh-wooley6x6	830	1792 (223)	3535 (2360)	3209 (1965)	-
b9	178	178	318 (161)	315 (160)	178
apex7	1660	1660	1221 (819)	980 (595)	1660
apex1	28336	28336	8901 (6002)	10545 (6930)	28336
alu4	1182	1182	2141 (1485)	2780 (1921)	1244
alu32r	189266	189266 (32)	18629 (12196)	18722 (12223)	-
alu32	12194	12194 (32)	959 (478)	959 (350)	-
alu2	231	231	420 (277)	433 (268)	233
adsb32r	528	624 (96)	897 (553)	867 (492)	-
adder16	327812	262310 (32)	606303 (376808)	589904 (360409)	-
C880	346660	346660	329484 (228241)	318161 (221408)	346660
C499	45922	7030 (32)	13699 (10074)	13704 (9829)	-
C432	1733	1733 (1)	4913 (3154)	4589 (2885)	-
C1908	36007	36007	37800 (26889)	37760 (26716)	36007
C1355	45922	45922	14162 (10538)	14167 (10293)	45922
rot	166674	166675 (3)	266795 (177820)	239984 (154832)	-
comp	458698	458698	859988 (495583)	819069 (360453)	-
Σ	**2.388.776**	**2.284.769**	**2.947.914**	**2.806.421**	n.a.
	100%	95.6	123.4%	117.5%	n.a.

Table A.2: Comparing OBDD and ⊕-OBDD **Size** for Fixed Variable Order - Combinatorial Circuits.

Circuit	OBDD-size		⊕-OBDD size				
			50%	20%	10%	5%	2,5%
sbc	3715	min	4302	3658	3562	**3531**	3700
		avg	4588	3987	3861	3806	3793
		max	5038	4219	4274	3961	3993
		σ	[5.1%]	[4.7%]	[5.6%]	[4.1%]	[2.4%]
s967	1732	min	**1336**	1549	1610	1763	1617
		avg	1585	1679	1705	1779	1742
		max	1855	1833	1797	1809	1787
		σ	[10.2%]	[5.9%]	[3.2%]	[0.8%]	[2.8%]
s820	2651	min	**664**	1365	1194	1762	2651
		avg	1839	2376	2480	2482	2673
		max	2943	3027	2993	2860	2798
		σ	[34.5%]	[21.5%]	[21.2%]	[15.7%]	[1.7%]
s713	**1352**	min	1803	1490	1381	1364	1352
		avg	2146	1680	1471	1482	1380
		max	2508	1936	1608	1659	1417
		σ	[10.5%]	[7.9%]	[4%]	[7.4%]	[1.5%]
s641	**1352**	min	1921	1421	1357	1360	1352
		avg	2194	1648	1506	1408	1371
		max	2553	1912	1678	1536	1396
		σ	[10.8%]	[8.8%]	[8.1%]	[4.1%]	[0.9%]
s635	656	min	**655**	655	656	655	656
		avg	659	656	656	655	656
		max	663	659	657	656	659
		σ	[0.4%]	[0.1%]	[0%]	[0.1%]	[0.1%]
s526	**232**	min	264	248	232	232	232
		avg	292	264	241	240	234
		max	313	290	254	262	244
		σ	[5.4%]	[4.5%]	[3.3%]	[4.1%]	[1.2%]
s510	19076	min	**704**	712	19016	19020	19076
		avg	6272	17251	19077	19071	19078
		max	19198	19119	19149	19095	19085
		σ	[141.7%]	[33.6%]	[0.2%]	[0.1%]	[0%]
s499	**336**	min	425	356	357	339	339
		avg	544	397	385	366	355
		max	602	441	424	386	372
		σ	[9.3%]	[6.2%]	[5.4%]	[3.8%]	[2.2%]
s444	226	min	285	233	228	**223**	226
		avg	331	274	244	237	232
		max	377	305	275	263	252
		σ	[8.4%]	[9.4%]	[6.1%]	[6.3%]	[4.3%]
s420	262227	min	**495**	139219	131472	131474	254754
		avg	107602	236890	210778	249153	261480
		max	254760	262244	262239	262236	262233
		σ	[92.5%]	[21.6%]	[30.7%]	[16.5%]	[0.9%]
s386	281	min	**275**	280	277	277	281
		avg	306	293	290	282	285
		max	330	324	304	287	296
		σ	[5.8%]	[4.4%]	[2.7%]	[1%]	[1.7%]
s3271	**3365**	min	4192	3708	3442	3376	3377
		avg	4728	4073	3715	3558	3515
		max	5835	4537	4327	4104	4104
		σ	[11.6%]	[6%]	[6.4%]	[7.2%]	[6.5%]

Table A.3: Influence of ⊕-Node Frequency on ⊕-OBDD Size - Complete Results (Part 1).

Circuit	OBDD-size		⊕-OBDD size				
			50%	20%	10%	5%	2,5%
s208	1033	min	**140**	624	623	624	1031
		avg	610	993	914	992	1033
		max	1043	1040	1039	1034	1038
		σ	[59.3%]	[12.9%]	[20.8%]	[13%]	[0.1%]
s1512	18896	min	**7114**	10662	10671	18915	18915
		avg	14371	17369	18163	18935	18961
		max	20562	19216	19133	18998	19134
		σ	[38.1%]	[20.3%]	[14.4%]	[0.1%]	[0.4%]
s1494	1016	min	1203	1100	1037	**1014**	1016
		avg	1284	1145	1094	1056	1037
		max	1415	1198	1160	1099	1087
		σ	[4.7%]	[3.1%]	[4.2%]	[2.5%]	[2.1%]
s1488	1016	min	1165	**1012**	1022	1016	1016
		avg	1230	1101	1050	1037	1022
		max	1292	1142	1090	1105	1064
		σ	[3.5%]	[3.8%]	[1.7%]	[2.6%]	[1.3%]
s1423	98454	min	**93909**	93909	97702	98557	97836
		avg	104907	100641	101243	100175	98919
		max	116530	106276	107074	104556	100470
		σ	[7.4%]	[4%]	[2.7%]	[1.9%]	[0.8%]
s1269	48176	min	39779	40707	**39197**	48233	48183
		avg	43637	47201	46811	48662	48350
		max	50851	51162	49286	49763	49078
		σ	[7.9%]	[6.9%]	[7.2%]	[0.9%]	[0.5%]
s1196	2294	min	2599	2582	2212	**2177**	2295
		avg	2855	2710	2416	2388	2338
		max	3055	3063	2613	2640	2429
		σ	[4.4%]	[5.4%]	[4.3%]	[6.2%]	[1.8%]
mm4a	675	min	1000	742	678	681	**674**
		avg	1197	914	859	775	749
		max	1306	1186	1027	1025	998
		σ	[8.2%]	[14.6%]	[16.7%]	[15%]	[16.9%]
dsip	13921	min	**13175**	13487	13676	13832	13873
		avg	13432	13683	13795	13869	13910
		max	13733	13858	13902	13933	13971
		σ	[1.3%]	[0.9%]	[0.5%]	[0.2%]	[0.2%]
x3	2760	min	1804	**1526**	2665	2789	2604
		avg	2799	2755	2836	2852	2775
		max	3365	3022	3020	2991	2862
		σ	[19.5%]	[16.1%]	[3.9%]	[2.3%]	[2.3%]
x1	**1297**	min	2170	1576	1450	1322	**1297**
		avg	2293	1764	1558	1425	1335
		max	2451	1973	1719	1531	1417
		σ	[4.5%]	[6.8%]	[5.5%]	[5.0%]	[4.0%]
vg2	1044	min	1458	1159	1046	1044	**1040**
		avg	1636	1279	1156	1090	1055
		max	1887	1541	1306	1156	1114
		σ	[8.2%]	[8.2%]	[7.2%]	[3.3%]	[1.9%]
vda	4345	min	**3092**	4105	4138	4002	4349
		avg	3743	4613	4520	4436	4401
		max	4742	5230	4940	4664	4528
		σ	[14.3%]	[7.1%]	[5.6%]	[4.2%]	[1.2%]

Table A.4: Influence of ⊕-Node Frequency on ⊕-OBDD Size - Complete Results (Part 2).

Circuit	OBDD-size		⊕-OBDD size				
			50%	20%	10%	5%	2,5%
too_large	**7096**	min	15521	11651	10229	8924	8384
		avg	17163	12697	10869	9405	8855
		max	18260	13552	11381	10015	9623
		σ	[4.6%]	[5%]	[3.5%]	[3.8%]	[4.5%]
term1	580	min	693	604	580	580	**558**
		avg	829	709	626	596	589
		max	959	877	768	674	672
		σ	[11.9%]	[12.9%]	[10.2%]	[4.8%]	[5%]
pair	**67685**	min	80092	71236	68739	68234	67923
		avg	88606	78894	71982	70609	69241
		max	103958	86911	82911	76428	72004
		σ	[8.1%]	[5%]	[5.9%]	[3.5%]	[2.6%]
my_adder	327677	min	297773	262222	262190	262192	**262188**
		avg	390205	310440	285503	292756	264034
		max	465113	369380	361218	348200	266310
		σ	[15.4%]	[11.9%]	[10.4%]	[10.9%]	[0.7%]
mux	131071	min	**217**	131071	131071	131071	131071
		avg	91814	131071	131071	131071	131071
		max	131071	131071	131071	131071	131071
		σ	[68.8%]	[0%]	[0%]	[0%]	[0%]
k2	**2760**	min	11819	21227	21667	25592	28001
		avg	19232	25625	27116	27192	28365
		max	24927	28661	29122	28896	28463
		σ	[21.7%]	[9.7%]	[9%]	[4.6%]	[0.4%]
i9	**2278**	min	5263	3431	2789	2506	2380
		avg	5757	3717	2992	2633	2451
		max	6215	3962	3367	2790	2564
		σ	[6%]	[4.3%]	[7%]	[3%]	[2.4%]
i8	**4366**	min	9853	6434	5382	4954	4449
		avg	10677	7152	5780	5266	4689
		max	11089	7686	6401	5661	5114
		σ	[3.3%]	[5.8%]	[4.7%]	[4.3%]	[3.7%]
i7	**505**	min	686	566	530	513	507
		avg	705	581	543	519	513
		max	745	602	556	525	527
		σ	[2.2%]	[1.8%]	[1.4%]	[0.5%]	[0.9%]
i5	**312**	min	434	345	316	**312**	**312**
		avg	473	372	334	330	315
		max	537	415	354	361	326
		σ	[6.9%]	[5.3%]	[3.2%]	[4.8%]	[1.5%]
i4	**421**	min	520	**421**	427	**421**	**421**
		avg	631	492	478	461	424
		max	748	619	518	512	445
		σ	[10.4%]	[11.3%]	[7.7%]	[6.5%]	[1.6%]
i2	335	min	335	335	335	335	335
		avg	645	335	386	335	335
		max	854	335	852	335	335
		σ	[41.2%]	[0%]	[42.2%]	[0%]	[0%]
frg2	6471	min	6583	6449	6559	**6092**	6445
		avg	7303	6939	6689	6512	6580
		max	7853	7324	6951	6749	6912
		σ	[6.3%]	[4%]	[1.7%]	[3.7%]	[2%]
frg1	**204**	min	248	220	204	204	204
		avg	272	233	215	209	206
		max	296	250	246	242	216
		σ	[5.5%]	[3.8%]	[5.1%]	[5.2%]	[1.9%]

Table A.5: Influence of ⊕-Node Frequency on ⊕-OBDD Size - Complete Results (Part 3).

Circuit	OBDD-size		⊕-OBDD size				
			50%	20%	10%	5%	2,5%
example2	469	min	479	467	**456**	466	468
		avg	499	483	475	475	472
		max	516	496	485	483	482
		σ	[2.2%]	[1.8%]	[1.6%]	[1%]	[0.8%]
count	**234**	min	320	249	239	**234**	**234**
		avg	430	296	282	250	234
		max	483	341	375	277	234
		σ	[11.3%]	[10.4%]	[14.5%]	[6.1%]	[0%]
cm150a	131071	min	**207**	207	207	131071	131071
		avg	39466	117984	117984	131071	131071
		max	131071	131071	131071	131071	131071
		σ	[160.1%]	[35%]	[35%]	[0%]	[0%]
booth8x8	6386	min	7486	6636	6120	5975	**5964**
		avg	10410	8489	6851	6380	6404
		max	12550	10752	9040	7252	7648
		σ	[18.7%]	[12.2%]	[14.5%]	[6.2%]	[9.4%]
bw6x6	**830**	min	2396	1855	1812	1795	1796
		avg	2555	2099	1951	1834	1842
		max	2812	2361	2131	1914	1985
		σ	[5.6%]	[6.2%]	[5.6%]	[1.9%]	[3.2%]
b9	**178**	min	197	194	178	178	178
		avg	232	221	198	183	181
		max	258	241	221	193	197
		σ	[7.7%]	[6.7%]	[6.5%]	[3.2%]	[3.3%]
apex7	1660	min	**1146**	1342	1363	1585	1510
		avg	1324	1538	1550	1625	1628
		max	1528	1692	1663	1679	1678
		σ	[8%]	[7.3%]	[5.2%]	[2.5%]	[3.3%]
apex1	28336	min	**15504**	24051	21372	21508	26536
		avg	23346	28374	26521	27338	28095
		max	28972	30908	30095	28740	28728
		σ	[19.6%]	[6.4%]	[10.6%]	[8.5%]	[2.7%]
alu4	1182	min	1618	1282	**1053**	1194	1184
		avg	1944	1560	1298	1340	1217
		max	2292	2022	1536	1740	1331
		σ	[11%]	[13.2%]	[10.6%]	[12.6%]	[4%]
alu32r	189266	min	**63974**	96157	153622	170621	162704
		avg	92939	146516	167093	182239	182294
		max	136123	174227	188298	189982	190099
		σ	[22.1%]	[14.9%]	[6.2%]	[3.9%]	[5.4%]
alu32	12194	min	**5192**	8003	10096	10799	11679
		avg	6877	9839	11022	11572	12004
		max	8403	10831	11886	12211	12321
		σ	[15.0%]	[8.0%]	[5.6%]	[3.6%]	[1.8%]
alu2	**231**	min	330	254	244	231	231
		avg	371	310	256	247	236
		max	404	393	275	281	244
		σ	[6.1%]	[16.4%]	[3.9%]	[6%]	[1.6%]
adsb32r	**528**	min	772	682	640	628	631
		avg	799	703	662	649	636
		max	827	746	692	667	654
		σ	[2.3%]	[3.1%]	[1.9%]	[1.6%]	[0.9%]
adder16	327812	min	320025	285085	262411	**262310**	262310
		avg	420763	342228	287074	275827	264098
		max	537425	455693	398660	356718	273939
		σ	[18.2%]	[16.9%]	[14.6%]	[10.4%]	[1.4%]

Table A.6: Influence of ⊕-Node Frequency on ⊕-OBDD Size - Complete Results (Part 4).

Circuit	OBDD-size		⊕-OBDD size				
			50%	20%	10%	5%	2,5%
C499	45922	min	7425	7150	7036	**7030**	7030
		avg	10190	7683	7846	7100	7080
		max	13254	8652	13308	7260	7272
		σ	[22.4%]	[7.2%]	[25%]	[1.2%]	[1.3%]
C432	**1733**	min	2457	1879	1765	1736	1733
		avg	3314	2473	2143	1906	1767
		max	4225	3415	3000	2087	1978
		σ	[13.6%]	[17.9%]	[17.4%]	[8%]	[4.2%]
C1908	36007	min	36319	37442	**35741**	35947	36007
		avg	40919	41304	39071	37861	36652
		max	49082	46944	45982	44697	38739
		σ	[9.4%]	[6.8%]	[7.5%]	[7%]	[2.3%]
C1355	45922	min	**23046**	35462	43336	42510	45922
		avg	32589	41617	45857	46232	46049
		max	42768	47310	49002	50684	47104
		σ	[18.5%]	[7.6%]	[4%]	[4.4%]	[0.8%]
bigkey	**6170**	min	8051	7157	6639	6417	6253
		avg	8139	7310	6770	6479	6322
		max	8225	7426	6862	6555	6369
		σ	[0.5%]	[1.1%]	[0.9%]	[0.6%]	[0.5%]
rot	166674	min	202102	170609	170856	167104	**163410**
		avg	217670	188105	178755	172378	167767
		max	235279	215030	189352	177577	175387
		σ	[4.7%]	[6.5%]	[3.1%]	[1.5%]	[1.8%]
mult16a	360442	min	408863	279404	265291	262190	**262188**
		avg	509929	330139	309691	311544	280296
		max	605804	376504	451340	451611	328478
		σ	[13.4%]	[10.4%]	[17.6%]	[22.2%]	[9.8%]
Σ	2.406.120	min	1.723.875	7.810.664	1.842.396	2.003.041	2.125.959
		avg	2.386.097	2.326.164	2.200748	2.254.635	2.186.692
		max	3.114.133	2.686.855	2.678.278	2.589.747	2.278.346

Table A.7: Influence of ⊕-Node Frequency on ⊕-OBDD Size - Complete Results (Part 5).

Circuit	OBDD	⊕-OBDD(⊕-nodes)						
		nDE	0.6	0.8	1.0	1.2	1.5	2.0
sbc	3715	4598	3757	3755	3792	3785	3775	3767
s967	1732	1073	1683	1683	1655	1642	1631	1640
s820	2651	552	1642	1642	1638	1634	2117	2651
s713	1352	3554	1433	1433	1408	1386	1366	1355
s641	1352	3550	1423	1423	1398	1376	1360	1355
s635	656	1746	798	793	660	660	659	655
s526	232	371	269	267	259	246	229	231
s510	19076	636	9738	9738	9742	9740	9713	9924
s499	336	640	570	570	337	337	337	356
s444	226	390	246	246	235	232	225	225
s420	262227	732	262334	262334	262311	262245	262227	262227
s386	281	295	273	255	262	260	271	280
s3271	3365	6437	4175	3865	3541	3498	3403	3369
s208	1033	186	1054	1054	1049	1049	1033	1033
s1512	18896	10941	18763	18762	18746	18727	18690	18729
s1494	1016	1378	1287	1287	1073	1078	1070	1015
s1488	1016	1316	1238	1200	1030	1033	1023	1015
s1423	98454	134008	99836	98531	98462	98460	98415	98519
s1269	48176	39922	50769	49966	49072	49068	48221	48211
s1196	2294	3844	2341	2341	2353	2341	2291	2263
rot	166674	266795	161506	163788	166825	166757	166705	166700
mult16a	360442	655125	163839	163839	163838	163838	163838	163838
mm9b	848081	658964	264856	264856	264856	264856	264838	264838
mm9a	735768	533707	220374	220374	220374	220374	220356	220356
comp	458698	859988	544543	544543	544541	544539	544549	544539
mm4a	675	1439	680	683	671	671	674	674
dsip	13921	9675	7722	13082	7717	7715	13923	13920
x3	2760	2369	2625	2495	2429	2428	2762	2761
x1	1297	2948	1358	1336	1331	1329	1307	1298
vg2	1044	2071	1120	1107	1050	1045	1042	1042
vda	4345	1954	4342	4048	4214	4293	4437	4344
too_large	7096	14507	7097	7097	7091	7090	7095	7095
term1	580	1161	590	586	584	584	579	579
pair	67685	108998	68085	68085	68006	68007	67983	68012
my_adder	327677	589831	196614	196613	196613	196613	196608	196608
mux	131071	217	131072	131072	131072	131072	131072	131070
k2	28336	5986	27474	27460	26361	27518	28137	28335
i9	2278	8754	3751	2277	2277	2277	2277	2277
i8	4366	14750	5208	4466	4433	4385	4365	4365
i7	505	1000	633	504	632	578	504	504
i5	312	763	523	460	523	451	329	317
i4	421	1095	430	430	430	430	428	420
i2	335	317	336	336	334	334	334	334
frg2	6471	5031	6235	6326	6348	6344	6354	6465
frg1	204	383	208	206	206	204	203	203
example2	469	644	457	473	456	456	454	454
count	234	294	226	235	226	226	227	225
cm150a	131071	220	131072	131072	131072	131072	131070	131070
bw6x6	830	3535	1733	1732	1714	1705	1693	1675
b9	178	318	204	190	198	196	182	179
apex7	1660	1221	1736	1569	1570	1556	1536	1659
apex1	28336	8901	38460	37389	26238	26314	28324	28337
alu4	1182	2141	1491	1641	1269	1265	1253	1243
alu32r	189266	18629	185397	185395	185394	185380	185361	188583
alu32	12194	959	8161	8161	8303	8686	10709	12193
alu2	231	420	297	240	237	234	232	232
adsb32r	528	897	698	688	688	687	685	623
adder16	327812	606303	458850	458850	262310	262308	262293	262293
C499	45922	13699	7093	7093	7029	7029	7029	7029
C432	1733	4913	1342	1342	1470	1736	1732	1732
C1908	36007	37800	41792	40971	35869	36013	36009	36007
C1355	45922	14162	47515	45921	45921	45921	45921	45921
bigkey	6170	7970	5518	5518	6188	6188	6176	6172
Σ	4.468.873	4.687.023	3.216.892	3.215.694	**2.997.931**	2.999.501	3.009.641	3.015.341
	100%	104.9%	72.0%	72.0%	**67.1%**	67.2%	67.3%	67.5%

Table A.8: Locally Greedy Heuristic for ⊕-Node Placement – (MAX/nDE).

Circuit	OBDD	⊕-OBDD						
		pDE	**0.6**	**0.8**	**1.0**	**1.2**	**1.5**	**2.0**
sbc	3715	5105	3819	3801	3781	3826	3803	3733
s967	1732	1208	1707	1708	1706	1710	1691	1676
s820	2651	718	1644	1644	1636	1634	2119	2651
s713	1352	3122	1504	1504	1486	1395	1366	1355
s641	1352	3089	1482	1482	1464	1375	1360	1355
s635	656	668	659	659	657	656	656	655
s526	232	342	248	246	242	238	229	231
s510	19076	739	9741	9741	9746	9738	9716	9926
s499	336	798	568	568	568	338	338	356
s444	226	399	245	245	241	234	224	225
s420	262227	471	262221	262221	262226	262226	262226	262226
s386	281	298	268	268	268	263	276	280
s3271	3365	6511	4348	3832	3613	3483	3397	3369
s208	1033	141	1031	1031	1032	1032	1032	1032
s1512	18896	7148	19031	19019	19004	18988	18969	18940
s1494	1016	1349	1148	1148	1111	1019	1061	1015
s1488	1016	1357	1138	1138	1050	1003	1003	1015
s1423	98454	113035	101073	98473	101349	98440	98377	98517
s1269	48176	33418	50324	49932	49214	49050	48220	48211
s1196	2294	3175	2179	2192	2212	2203	2173	2179
rot	166674	240643	161550	163861	163973	166724	166679	166682
mult16a	360442	745320	163838	163838	163838	163838	163838	163838
mm9b	848081	61233	678306	515849	264918	264856	264838	264838
mm9a	735768	50926	220384	220384	220375	220375	220356	220356
comp	458698	544533	544533	544541	544541	544539	544539	544539
mm4a	675	1407	687	687	674	672	674	674
dsip	13921	12363	7724	7719	7719	7715	7716	13920
x3	2760	2230	2683	2683	2511	2436	2761	2761
x1	1297	2978	1400	1400	1387	1380	1325	1309
vg2	1044	2195	1119	1119	1097	1044	1042	1043
vda	4345	1895	3868	4116	4096	4031	4268	4344
too_large	7096	21858	7096	7096	7090	7090	7095	7095
term1	580	1059	583	583	582	582	579	579
pair	67685	114057	68128	68114	68102	68034	67992	68013
my_adder	327677	524297	196614	196613	196613	196613	196608	196608
mux	131071	217	131072	131072	131072	131072	131072	131072
k2	28336	7479	29410	29411	28356	27390	27823	28335
i9	2278	10258	3751	3633	2277	2277	2277	2277
i8	4366	15307	6473	5336	4641	4580	4365	4365
i7	505	889	633	633	632	578	504	504
i5	312	636	399	399	399	362	329	317
i4	421	845	424	424	424	424	424	420
i2	335	851	336	336	336	334	334	334
frg2	6471	7246	6202	6196	6358	6364	6364	6462
frg1	204	334	208	208	206	204	203	203
example2	469	485	454	454	454	454	456	454
count	234	308	307	307	307	230	227	225
cm150a	131071	207	131072	131072	131072	131072	131070	131070
bw6x6	830	3209	1652	1648	1635	1630	1628	1641
b9	178	315	200	200	200	192	184	179
apex7	1660	980	1788	1711	1560	1540	1536	1659
apex1	28336	10545	38937	38889	37667	27643	27974	28335
alu4	1182	2780	1678	1342	1269	1262	1252	1243
alu32r	189266	18722	185410	185398	185394	185290	185268	186902
alu32	12194	959	8161	8161	8161	8161	8161	12067
alu2	231	433	289	289	238	234	232	232
adsb32r	528	867	693	688	687	687	686	623
adder16	327812	589904	458850	458850	458850	262398	262398	262293
C499	45922	13704	7093	7093	7029	7029	7029	7029
C432	1733	4589	1340	1340	1337	1734	1732	1732
C1908	36007	37760	41795	41787	41767	36008	36008	36007
C1355	45922	14167	47419	47560	45937	45921	45921	45921
bigkey	6170	7633	5519	5519	5518	6188	6176	6172
Σ	4.468.873	3.261.714	3.634.456	3.469.411	3.213.905	**3.000.038**	3.000.179	3.013.619
	100%	73%	81.3%	77.6%	71.9%	**67.1%**	67.1%	67.4%

Table A.9: Locally Greedy Heuristic for ⊕-Node Placement – (MAX/pDE).

Circuit	OBDD	⊕-OBDD(⊕-nodes)						
		nDE	0.6	0.8	1.0	1.2	1.5	2.0
sbc	3715	4598	3755	3733	3752	3779	3765	3727
s967	1732	1073	1682	1682	1664	1654	1640	1754
s820	2651	552	1642	1635	1628	1626	2653	2651
s713	1352	3554	1431	1425	1401	1367	1367	1351
s641	1352	3550	1423	1415	1391	1361	1361	1351
s635	656	1746	693	692	655	655	655	655
s526	232	371	265	255	231	231	231	231
s510	19076	636	9738	9721	9863	9882	9924	9924
s499	336	640	562	338	356	356	356	356
s444	226	390	239	225	225	225	225	225
s420	262227	732	262332	262322	262227	262227	262227	262227
s386	281	295	257	262	272	280	280	280
s3271	3365	6437	3799	3661	3400	3392	3368	3365
s208	1033	186	1052	1050	1033	1033	1033	1032
s1512	18896	10941	18758	18745	18683	18694	18733	18911
s1494	1016	1378	1249	1200	1085	1062	1015	1015
s1488	1016	1316	1200	1151	1023	1019	1015	1015
s1423	98454	134008	101394	101384	101506	98589	98556	98545
s1269	48176	39922	49925	49671	49042	49013	48180	48176
s1196	2294	3844	2342	2323	2305	2451	2267	2294
rot	166674	266795	161507	163747	163832	166700	166678	166685
mult16a	360442	655125	163839	163838	163838	163838	163838	163838
mm9b	848081	658964	264856	264838	264838	264838	264838	264838
mm9a	735768	533707	220374	220356	220356	220356	220356	220356
comp	458698	859988	544541	544541	544539	544539	544539	544539
mm4a	675	1439	680	683	674	674	674	674
dsip	13921	9675	7719	13082	13887	13920	13920	13920
x3	2760	2369	2511	2495	2394	2390	2761	2761
x1	1297	2948	1349	1336	1308	1300	1298	1296
vg2	1044	2071	1120	1107	1050	1046	1042	1043
vda	4345	1954	4268	4048	4102	4291	4496	4344
too_large	7096	14507	7097	7097	7090	7090	7095	7095
term1	580	1161	590	586	579	579	579	579
pair	67685	108998	68069	68046	67995	67979	67976	67960
my_adder	327677	589831	196613	196610	196610	196610	196608	196608
mux	131071	217	131072	131072	131072	131072	131072	131070
k2	28336	5986	27474	27460	26492	28021	28503	28335
i9	2278	8754	3633	2277	2277	2277	2277	2277
i8	4366	14750	5119	4466	4365	4365	4365	4365
i7	505	1000	632	504	504	504	504	504
i5	312	763	523	460	327	311	311	311
i4	421	1095	430	430	428	428	420	420
i2	335	317	336	336	334	334	334	334
frg2	6471	5031	6379	6356	6350	6342	6341	6465
frg1	204	383	208	206	203	203	203	203
example2	469	644	456	456	453	452	452	452
count	234	294	226	227	227	225	225	225
cm150a	131071	220	131072	131072	131072	131072	131070	131070
bw6x6	830	3535	1732	1700	1707	1661	1661	1659
b9	178	318	204	190	182	179	179	177
apex7	1660	1221	1657	1569	1544	1532	1532	1659
apex1	28336	8901	37596	37389	26345	26956	28228	28336
alu4	1182	2141	1783	1659	1251	1243	1243	1243
alu32r	189266	18629	185392	185391	185551	185493	185513	189295
alu32	12194	959	8161	8184	10652	11378	11913	12193
alu2	231	420	296	242	234	232	232	232
adsb32r	528	897	691	687	623	623	623	623
adder16	327812	606303	458850	458851	262293	262293	262293	262293
C499	45922	13699	7093	7093	7029	7029	7029	7029
C432	1733	4913	1339	1614	1732	1732	1732	1732
C1908	36007	37800	41760	40971	38316	36009	36009	36006
C1355	45922	14162	45921	45921	45921	45921	45921	45921
bigkey	6170	7970	5518	6170	6172	6172	6172	6172
Σ	4.468.873	4.687.023	3.214.424	3.218.253	3.008.490	3.009.105	3.011.906	3.016.222
	100%	104.9%	71.9%	72.0%	**67.3%**	67.3%	67.4%	67.5%

Table A.10: Locally Greedy Heuristic for ⊕-Node Placement – (ADD/nDE).

Circuit	OBDD	⊕-OBDD(⊕-nodes)					
		MAX		ADD		MAX	
		nDE-first	pDE-first	nDE-first	pDE-first	nDE	pDE
sbc	3715	4503	4953	4551	5022	3792	3781
s967	1732	1083	1206	1083	1197	1655	1706
s820	2651	558	710	572	712	1638	1636
s713	1352	3504	3083	3527	3119	1408	1486
s641	1352	3497	3039	3520	3073	1398	1464
s635	656	1672	690	1745	667	660	657
s526	232	342	321	368	341	259	242
s510	19076	617	684	619	699	9742	9746
s499	336	404	585	639	797	337	568
s444	226	382	374	389	398	235	241
s420	262227	698	469	731	470	262311	262226
s386	281	311	309	299	304	262	268
s3271	3365	6180	6218	6434	6510	3541	3613
s208	1033	170	139	185	140	1049	1032
s1512	18896	11042	7069	10940	7147	18746	19004
s1494	1016	1368	1290	1400	1297	1073	1111
s1488	1016	1288	1284	1321	1297	1030	1050
s1423	98454	134283	113365	133993	113363	98462	101349
s1269	48176	39831	33337	39875	33375	49072	49214
s1196	2294	3918	3409	3958	3251	2353	2212
bigkey	6170	8888	8268	7969	7632	6188	5518
dsip	13921	11235	13704	9674	12362	7717	7719
mm4a	675	1429	1394	1438	1406	671	674
mm9a	735768	533688	550714	533706	550741	220374	220375
mm9b	848081	658945	629337	658963	629363	264856	264918
mult16a	360442	376795	376794	376796	376794	163838	163838
x3	2760	2171	2011	2364	2141	2429	2511
x1	1297	2894	2856	2933	2949	1331	1387
vg2	1044	2054	2127	2063	2133	1050	1097
vda	4345	2106	1920	1965	1906	4214	4096
too_large	7096	14500	21852	14506	21858	7091	7090
term1	580	1059	1052	1159	1058	584	582
pair	67685	108813	113859	108862	113926	68006	68102
my_adder	426088	393209	327675	393212	327678	196613	196613
mux	131071	216	214	216	216	131072	131072
k2	28336	6127	7530	5997	7503	26361	28356
i9	2278	5504	7047	5789	7347	2277	2277
i8	4366	10646	14652	14624	15306	4433	4641
i7	505	870	759	999	888	632	632
i5	312	546	545	716	619	523	399
i4	421	1084	844	1086	844	430	424
i2	335	316	848	316	850	334	336
frg2	6471	5541	8302	5113	7393	6348	6358
frg1	204	377	328	382	333	206	206
example2	469	638	504	643	508	456	454
count	234	308	521	308	572	226	307
cm150a	131071	217	204	219	206	131072	131072
bw6x6	830	3360	3305	3408	2997	1714	1635
b9	178	281	285	312	308	198	200
apex7	1660	1106	891	1207	967	1570	1560
apex1	28336	8702	10454	8527	10362	26238	37667
alu4	1182	2135	2196	2137	2211	1269	1269
alu32r	189266	18785	14572	18628	18782	185394	185394
alu32	12194	1088	682	958	958	8303	8161
alu2	231	378	358	419	362	237	238
adsb32r	528	804	774	868	837	688	687
adder16	327812	606407	590010	606286	589886	262310	458850
C499	45922	13634	13639	13698	13671	7029	7029
C432	1733	3925	3653	4912	4588	1470	1337
C1908	36007	38002	38003	37842	37806	35869	41767
C1355	45922	14562	14567	14161	14166	45921	45937
comp	458698	859981	819062	859985	819066	544541	544541
rot	166674	264315	239836	266731	239946	166825	163973
Σ	**4.468.873**	**4.203.380**	**4.030.682**	**4.208.246**	**4.034.624**	**2.997.931**	**3.213.905**
	100%	94.1%	90.2 %	94.2%	90.3 %	67.1%	71.9%

Table A.11: Locally Greedy Heuristic for ⊕-Node Placement – (nDE/pDE first).

Circuit	⊕-OBDD size [Bytes]	
	binary ⊕-nodes	meta-⊕-nodes
sbc	165528	135332 [82%]
s967	38628	27304 [71%]
s820	19872	16532 [83%]
s713	127944	115428 [90%]
s641	127800	115360 [90%]
s635	62856	64692 [102%]
s526	13356	10836 [81%]
s510	22896	15444 [67%]
s499	23040	24128 [104%]
s444	14040	11136 [79%]
s420	26352	19932 [75%]
s386	10620	9900 [93%]
s3271	231732	189228 [81%]
s208	6696	5092 [76%]
s1512	393876	330636 [83%]
s1494	49608	36116 [72%]
s1488	47376	34332 [72%]
s1423	4824288	3935156 [81%]
s1269	1437192	1101680 [76%]
s1196	138384	118808 [85%]
bigkey	286920	203712 [70%]
dsip	348300	292340 [83%]
mm4a	51804	34256 [66%]
mult16a	23584500	23852212 [101%]
mm9b	23722704	18157936 [76%]
mm9a	19213452	14583716 [75%]
Σ	**74.989.764**	**63.437.244 [84.6%]**

Table A.12: ⊕-OBDD Size for Binary ⊕-Nodes vs. Meta-⊕-Nodes (Part 1) – Sequential Circuits.

Circuit	⊕-OBDD size [Bytes]	
	binary ⊕-nodes	meta-⊕-nodes
x3	85284	76072 [89%]
x1	106128	81780 [77%]
vg2	74556	66084 [88%]
vda	70344	43336 [61%]
too_large	522252	423484 [81%]
term1	41796	34192 [81%]
pair	3923928	3409992 [86%]
my_adder	21233916	18873336 [88%]
mux	7812	1960 [25%]
k2	215496	161348 [74%]
i9	315144	236480 [75%]
i8	531000	404896 [76%]
i7	36000	31172 [86%]
i5	27468	21660 [78%]
i4	39420	37320 [94%]
i2	11412	10880 [95%]
frg2	181116	139004 [76%]
frg1	13788	11684 [84%]
example2	23184	21168 [91%]
count	10584	10068 [95%]
cm150a	7920	1960 [24%]
booth8x8	520416	444428 [85%]
baugh-wooley6x6	127260	95832 [75%]
b9	11448	10700 [93%]
apex7	43956	27720 [63%]
apex1	320436	197496 [61%]
alu4	77076	53692 [69%]
alu32r	670644	460920 [68%]
alu32	34524	31628 [91%]
alu2	15120	9856 [65%]
adsb32r	32292	47448 [146%]
adder16	21826908	21629848 [99%]
C880	11861424	8576036 [72%]
C499	493164	670188 [135%]
C432	176868	184040 [104%]
C1908	1360800	1162344 [85%]
C1355	509832	670024 [131%]
rot	9604620	9197664 [95%]
comp	30959568	27142104 [87%]
Σ	**106.124.904**	**94.709.844 [89.2%]**

Table A.13: ⊕-OBDD Size for Binary ⊕-Nodes vs. Meta-⊕-Nodes (Part 2) – Combinatorial Circuits.

Circuit	⊕-OBDD size [Bytes]	
	binary ⊕-nodes	meta-⊕-nodes
sbc	165528	135148 [81%]
s967	38628	27268 [70%]
s820	19872	16660 [83%]
s713	127944	115808 [90%]
s641	127800	115740 [90%]
s635	62856	69032 [109%]
s526	13356	10836 [81%]
s510	22896	15380 [67%]
s499	23040	24128 [104%]
s444	14040	11092 [79%]
s420	26352	19869 [75%]
s386	10620	9244 [87%]
s3271	231732	187596 [80%]
s208	6696	5092 [76%]
s1512	393876	330064 [83%]
s1494	49608	35668 [71%]
s1488	47376	33240 [70%]
s1423	4824288	3913248 [81%]
s1269	1437192	988904 [68%]
s1196	138384	117684 [85%]
bigkey	286920	203712 [70%]
dsip	348300	292340 [83%]
mm4a	51804	34020 [65%]
mm9b	23722704	18157936[76%]
mm9a	19213452	14583716 [75%
mult16a	23584500	23851068 [101%]
Σ	**74.989.764**	**63.304.493 [84.4%]**

Table A.14: ⊕-OBDD Size for Synthesis with Meta-⊕-Nodes (Part 1) – Sequential Circuits.

Circuit	⊕-OBDD size [Bytes]	
	binary ⊕-nodes	meta-⊕-nodes
x3	85284	75992 [89%]
x1	106128	81780 [77%]
vg2	74556	65028 [87%]
vda	70344	43572 [61%]
too_large	522252	412176 [78%]
term1	41796	33400 [79%]
pair	3923928	3407336 [86%]
my_adder	21233916	18873336 [88%]
mux	7812	1960 [25%]
k2	215496	186864 [87%]
i9	315144	236480 [75%]
i8	531000	403068 [75%]
i7	36000	31324 [87%]
i5	27468	21660 [78%]
i4	39420	37320 [94%]
i2	11412	10880 [95%]
frg2	181116	135120 [74%]
frg1	13788	11684 [84%]
example2	23184	20924 [90%]
count	10584	10068 [95%]
cm150a	7920	1960 [25%]
booth8x8	520416	443264 [85%]
baugh-wooley6x6	127260	94728 [74%]
b9	11448	10700 [93%]
apex7	43956	26748 [60%]
apex1	320436	363867 [114%]
alu4	77076	53368 [69%]
alu32r	670644	460920 [68%]
alu32	34524	31628 [91%]
alu2	15120	9812 [64%]
adsb32r	32292	47448 [146%]
adder16	21826908	21629884 [99%]
C880	11861424	8603272 [73%]
C499	493164	670188 [135%]
C432	176868	187656 [106%]
C1908	1360800	1161388 [85%]
C1355	509832	670024 [131%]
rot	9604620	9197664 [95%]
comp	30959568	27142104 [87%]
Σ	**106.124.904**	**94.906.595 [89.4%]**

Table A.15: ⊕-OBDD Size for Synthesis with Meta-⊕-Nodes (Part 2) – Combinatorial Circuits.

Circuit		⊕-OBDD size [Bytes]			
	OBDD	pDE-meta	Jiggle (1)	Jiggle (2)	Jiggle (3)
sbc	133740	135892	117584	117220	-
s967	62352	27464	23612	23564	-
s820	95436	16572	13856	13460	-
s713	48672	115788	98272	98128	-
s641	48672	115720	98244	98100	-
s635	23616	64692	42688	42688	-
s526	8352	10836	9324	9324	-
s510	686736	15484	12388	11872	-
s499	12096	24128	12964	-	-
s444	8136	11416	8424	-	-
s420	9440172	19932	16436	16412	-
s386	10116	9900	8552	-	-
s3271	121140	189548	141100	-	-
s208	37188	5092	4448	4436	-
s1512	680256	331236	300732	300732	-
s1494	36576	36196	31068	29688	-
s1488	36576	34492	30092	29584	29296
s1423	3544344	3938476	3360020	3341752	-
s1269	1734336	1101800	965772	964792	963868
s1196	82584	118968	102108	101800	101776
mm4a	24300	34376	29632	29180	28412
dsip	501156	292420	254988	238376	-
bigkey	222120	203712	184552	-	-
8085	4242276	2614032	2255804	2253144	2253004
Σ	**21.840.948**	**9.468.172**	**8.123.556**	**8.080.368**	**8.078.044**
	230.7%	100.0 %	85.7 %	85.3 %	85.3 %
not finished					
mm9a	26487648	21382240	-	-	-
mm9b	30530916	24422348	-	-	-
mult16a	12975912	23852212	-	-	-

Table A.16: Jiggle Heuristic for ⊕-node Placement – Sequential Circuits.

Circuit		⊕-OBDD size [Bytes]			
	OBDD	**pDE-meta**	**Jiggle (1)**	**Jiggle (2)**	**Jiggle (3)**
x3	99360	76232	64520	-	-
x1	46692	81900	46972	-	-
vg2	37584	66124	39140	-	-
vda	156420	43416	35096	31028	-
too_large	255456	423524	371800	369712	369580
term1	20880	34232	24864	19704	-
pair	2436660	3411992	3100748	3100316	-
my_adder	11796372	18873336	16252724	13369124	-
mux	4718556	2240	1896	-	-
k2	1020096	161748	150268	149432	-
i9	82008	236480	82896	-	-
i8	157176	404896	283212	202136	-
i7	18180	31172	18240	-	-
i5	11232	21660	12672	11196	-
i4	15156	37320	22920	15164	-
i2	12060	10880	10396	-	-
frg2	232956	142604	119036	-	-
frg1	7344	11684	7324	-	-
example2	16884	21288	16132	-	-
count	8424	10068	8836	-	-
cm150a	4718556	2280	1876	-	-
booth8x8	229896	445108	444060	437500	437464
baugh-wooley6x6	29880	96592	87752	86652	-
b9	6408	10740	6744	-	-
apex7	59760	29080	23476	-	-
apex1	1020096	197656	187108	185165	-
alu4	42552	53772	40064	38108	34448
alu32r	6813576	460920	334264	-	-
alu32	438984	31628	26384	-	-
alu2	8316	9856	7928	-	-
adsb32r	19008	47448	41624	38072	-
adder16	11801232	21629848	21238784	-	-
C499	1653192	670188	671680	670072	-
C432	62388	184080	83044	-	-
C1908	1296252	1163384	1129004	1109496	1108708
C1355	1653192	670024	671516	669908	-
Σ	**51.002.784**	**49.805.400**	**45.665.000**	**42.638.916**	**42.634.300**
	102.4 %	100.0 %	91.6 %	85.6 %	85.6 %
not finished					
C880	12479760	8576876	-	-	-
comp	16513128	27142104	-	-	-
rot	6200064	9197664	-	-	-

Table A.17: Jiggle Heuristic for ⊕-node Placement – Combinatorial Circuits.

Circuit		⊕-OBDD size [Bytes]			
	OBDD	pDE-DynJiggle	Jiggle (1)	Jiggle (2)	Jiggle (3)
sbc	133740	135452	120152	119400	-
s967	38628	27568	23728	23716	-
s820	95436	16908	14612	14108	-
s713	48672	114980	98396	98216	-
s641	48672	114912	97412	97376	-
s635	23616	69032	46092	-	-
s526	8352	10836	9196	-	-
s510	686736	15444	12016	11696	-
s499	12096	24128	12964	-	-
s444	8136	11136	10292	-	-
s420	9440172	19932	17812	-	-
s386	10116	9900	8316	-	-
s3271	121140	189184	169292	-	-
s208	37188	5092	4460	-	-
s1512	680256	330636	303736	303596	-
s1494	36576	36068	32028	31856	-
s1488	36576	34216	28772	28532	-
s1423	3544344	3937408	3622180	3561860	-
s1269	1734336	1106220	1009468	1008068	1007172
s1196	82584	118484	102980	103004	-
8085	4242276	4376740	4242916	4187524	-
bigkey	222120	203712	184052	-	-
dsip	501156	292340	254172	245360	-
mm4a	24300	34256	28728	-	-
mm9a	26487648	21406700	21405004	-	-
mm9b	30540926	24485432	24482980	-	-
mult16a	12975912	23822712	19286296	-	-
Σ	**91.821.710**	**80.949.428**	**75.628.052**	**75.500.380**	**75.494.204**
	100.0 %	88.2 %	82.4 %	82.2 %	82.2 %

Table A.18: Dynamic Application of the Jiggle Heuristic – Sequential Circuits.

Circuit		⊕-OBDD size [Bytes]			
	OBDD	pDE-DynJiggle	Jiggle (1)	Jiggle (2)	Jiggle (3)
x3	99360	77636	71184	71064	-
x1	46692	81780	75416	75456	-
vg2	37584	66084	53360	52152	51828
vda	156420	44168	34556	33968	-
too_large	522252	422884	421724	420892	-
term1	20880	34192	24732	-	-
pair	2436660	3410052	3153832	3147496	3147436
my_adder	11796372	18854096	14486324	14486056	-
mux	4718556	1960	1896	-	-
k2	1020096	161892	150268	149432	-
i9	82008	236480	214536	-	-
i8	157176	404736	283040	283804	-
i7	18180	31324	21912	-	-
i5	11232	21660	12672	12396	-
i4	15156	37320	23864	23704	-
i2	12060	10880	10396	-	-
frg2	232956	139140	120532	120292	-
frg1	7344	11684	12716	-	-
example2	16884	21168	18152	-	-
count	8424	10068	8836	-	-
cm150a	4718556	1960	1876	1876	-
booth8x8	229896	444376	445116	441412	441048
baugh-wooley6x6	29880	95752	90616	90252	-
b9	6408	10700	8996	-	-
apex7	59760	27720	23268	-	-
apex1	1020096	201152	182324	-	-
alu4	42552	53740	38904	38712	-
alu32r	6813576	460920	364968	-	-
alu32	438984	31628	28464	-	-
alu2	8316	9984	8712	8260	-
adsb32r	19008	47448	39692	-	-
adder16	11801232	21467152	20836304	20823676	-
C880	12456	10571900	10291912	10170540	10085944
C499	16524	670188	671924	670388	-
C432	62388	186136	175592	-	-
C1908	1296252	1162736	1240312	-	-
C1355	1653192	670024	921260	897168	894744
comp	16513128	19746056	19663748	18741360	18741216
rot	36166674	7705076	7112688	6918432	-
Σ	**102.325.170**	**87.643.852**	**81.346.724**	**80.055.460**	**79.967.320**
	100.0 %	85.7 %	79.5 %	78.2 %	78.2 %

Table A.19: Dynamic Application of the Jiggle Heuristic - Combinatorial Circuits.

Circuit	OBDD-size	⊕-OBDD size [Bytes]		
	Sifting	pDE-meta	Sifting	+ Dynamic Jiggle
sbc	38052	135332	38556	35672
s967	14184	27304	16156	14652
s820	8172	16532	7044	6264
s713	22644	115428	18408	17888
s641	22644	115360	16044	14660
s635	4716	64692	10364	3356
s526	5076	10836	4344	3896
s510	5904	15444	6684	5840
s499	11880	24128	12952	12600
s444	5796	11136	1588	1340
s420.1	2916	19932	3376	2708
s386	4428	9900	3972	3708
s3271	31104	189228	41156	29908
s208	2196	5092	2308	2088
s1512	28224	330636	16072	14280
s1494	14076	36116	14652	12888
s1488	14076	34332	14992	13356
s1423	64656	3935156	55580	54468
s1269	78804	1101680	79140	76328
s1196	23076	118808	21984	19384
bigkey	140292	203712	141432	140192
dsip	88416	292340	108468	89436
mm4a	8604	34256	8168	8244
mm9b	90936	18157936	74108	73292
mm9a	72828	14583716	57264	56472
mult16a	7380	23852212	8032	8032
Σ	**811.080**	**63.437.244**	**782.844**	**720.952**
	100%	7821%	96.5%	88.9%
s007	271152	-	275552	274628
s838.1	6444	-	6516	5984
s3384	32616	-	40744	32432
s13207.1	108288	-	97444	96760
s13850.1	451404	-	415404	404588
Σ	**1.680.984**	**-**	**1.618.504**	**1.535.344**
	100%		**96.3%**	**91.3%**

Table A.20: Dynamic Application of the Sifting Heuristic - Sequential Circuits.

Circuit	OBDD-size	⊕-OBDD size [Bytes]		
	Sifting	pDE-meta	Sifting	+ Dynamic Jiggle
x3	20628	76072	21468	20088
x1	17244	81780	18020	16972
vg2	8244	66084	8516	8208
vda	17892	43336	18748	17368
too_large	20484	522252	20552	20448
term1	5868	34192	6200	5760
pair	128664	3409992	106776	106376
my_adder	4716	18873336	5868	5100
mux	1188	1960	1288	1152
k2	45972	161348	47124	45160
i9	51840	236480	52840	51812
i8	48348	236480	50092	48488
i7	14148	31172	20800	13928
i5	4824	21660	4788	4788
i4	8928	37320	9360	8568
i2	7380	10880	7344	7344
frg2	51624	139004	53136	50624
frg1	3348	11684	3436	3312
example2	10620	21168	12140	8960
count	2916	10068	3196	3152
cm150a	1188	1960	1240	1152
booth8x8	220320	444428	256192	254688
bw6x6	28980	95832	67940	67628
b9	3960	10700	4756	3852
apex7	10944	27720	11576	9392
apex1	45864	197496	45912	44872
alu4	21708	53692	21904	21724
alu32r	61956	460920	64736	61920
alu32	17208	31628	19988	17172
alu2	5832	9856	5768	5760
adsb32r	12420	47448	13444	13356
adder16	8136	21629848	13520	11128
C880	375840	8576036	358096	344160
C499	958464	670188	196708	187892
C432	43560	184040	37160	36792
C1908	230220	1162344	231412	230112
C1355	1064232	670024	1065340	1064196
rot	215028	9197664	224408	209332
comp	4608	27142104	3928	3796
Σ	**3.811.344**	**94.709.844**	**3.115.020**	**3.052.0932**
	100%	2485%	81.7 %	80.1%
dpath32	23112	-	48140	47688
dalu	29916	-	31332	30012
alu128	1695960	-	1707188	1707188
add128r	75672	-	83840	78548
i10	2446956	-	1906792	1904572
mm30a	3621276	-	3010564	3009244
C3540	862200	-	862076	862076
C5315	66384	-	62760	53780
C7552	296674	-	254196	254196
Σ	**12.020.404**	n.a.	**11.082.232**	**11.000.212**
	100%	n.a.	**92.2%**	**91.5%**

Table A.21: Dynamic Application of the Sifting Heuristic - Combinatorial Circuits.

Bibliography

[ABH+86] M. Ajtai, L. Babai, P. Hajnal, J. Komlos, P. Pudlak, V. Rödl, E. Szemeredi, G. Turan, Two Lower Bounds for Branching Programs, *in Proc. of 18th ACM Symposium on the Theory of Computing (STOC)*, 1986, 30-38.

[Ake78] S. B. Akers, Binary Decision Diagrams, *in IEEE Transactions on Computers*, vol. **c-27**, no. 6, 1978, 509-516.

[BC95] R. E. Bryant, Y.-A. Chen, Verification of Arithmetic Circuits with Binary Momentum Diagrams, *in Proc. 32nd ACM/IEEE Design Automation Conference (Santa Clara, CA)*, 1995, 535-541.

[BCW80] M. Blum, A. K. Chandra, M N. Wegman, Equivalence of Free Boolean Graphs can be decided Probabilistically in Polynomial Time, *in Information Processing letters* **10**, No. 2, 1980, 80-82.

[BD95] B. Becker, R. Drechsler, How many Decomposition Types do we need?, *in Proc. IEEE European Design and Test Conference*, 1995, 438-443.

[BDT97] B. Becker, R. Drechsler, M. Theobald, On the Expressive Power of OKFDDs, *in Formal Methods in System Design* **11**, 1997, 5-21.

[BDW95] B. Becker, R. Drechsler, R. Werchner, On the Relation between BDDs and FDDs, *in Information and Computation* **123(2)**, 1995, 185-197.

[BFG+97] I. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, F. Somenzi, Algebraic Decision Diagrams and their Applications, *in Journal of Formal Methods in System Design* **10**(2/3), 1997, 171-206.

[BHR95] Y. Breitbart, H. B. Hunt III, D. Rosenkrantz, On the Size of Binary Decision Diagrams Representing Boolean Functions, *Theoretical Computer Science* **145**, 1995, 45-69.

[BHS+96] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincenteli, F. Somenzi, et al., VIS: A System for Verification and Synthesis, *in Proc. Computer-Aided Verification'96*, vol 1102 of *Lecture Notes in Computer Science*, Springer, 1996, 428-432.

[BLS+95] B. Bollig, M. Löbbing, D. Sieling, I. Wegener, Complexity Theoretical Aspects of OFDDs, *in Proc. of IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design (Chiba, Japan)*, 1995, 198-205.

[BLS+99] B. Bollig, M. Löbbing, D. Sieling, I. Wegener, On the Complexity of the Hidden Weighted Bit Function for Various BDD Models, *in Theoretical Informatics and Applications* **33**, 1999, 103-115.

[BM77] G. Birkhoff, S. MacLane, Brief Survey of Modern Algebra, 4th ed., *MacMillan, New York*, 1977.

[Bra92] K. Brace, Ordered Binary Decision Diagrams for Optimization in Symbolic Switch-Level Analysis of MOS Circuits, *PhD Thesis*, Dept. of Electrical and Computer Engeneering, Carnegie Mellon University, Pittsburgh, PA, 1992.

[Bry86] R. E. Bryant, Graph-based Algorithm for Boolean Function Manipulation, *in IEEE Transactions on Computers* **C-35** No. 8, 1986, 677-691.

[Bry91] R. E. Bryant, On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Applications to Integer Multiplication, *in IEEE Transactions on Computers* **40** Vol. 2, 1991, 205-213.

[Bry92] R. E. Bryant, Symbolic Manipulation with Ordered Binary Decision Diagrams, *in ACM Computing Surveys* **24** Vol. 3, 1992, 293-318.

[BRB90] K. S. Brace, R. L. Rudell, R. E. Bryant, Efficient Implementation of a BDD Package, *in Proc. of the 27th ACM/IEEE Design Automation Conference*, 1990, 40-45.

[BSS+98] B. Bollig, M. Sauerhoff, D. Sieling, I. Wegener, Hierarchy Theorems for kOBDDs and kIBDDs, *in Theoretical Computer Science* **205**, 1998, 45-60.

[Bur91] J. R. Burch, Using BDDs to Verify Multipliers, *in Proc. of the 28th ACM/IEEE Design Automation Conference*, 1991, 408-412.

[BW96] B. Bollig, I. Wegener, Improving the Variable Ordering of OBDDs in NP-complete, *in IEEE Transactions on Computers* **45**, 1996, 993-1002.

[CHS74] R. L. Constable, H. B. Hunt III, S. Sahni, On the Computational Complexity of Scheme Equivalence, *in Proc. 8th Princeton Conference on Information Sciences and Systems*, 1974.

[CFM+93] E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, X. Zhao, Multi Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation, *in Proc. of the International Workshop on Logic Synthesis*, 1993, 1-15.

[DS00] E. Dubrova, H. Sack, Probabilistic Verification of Multiple-Valued Functions, *in Proc. of 30th IEEE International Symposium on Multiple-Valued-Logic (ISMVL 2000), (Portland, Oregon, USA)*, 2000, 460-466.

[DDG98] R. Drechsler, N. Drechsler, W. Günther, Fast Exact Minimization of BDDs, *in Proc. 35th ACM/IEEE Design Automation Conference (San Francisco, CA)*, 1998, 200-205.

[DG99] R. Drechsler, W. Günther, Using Lower Bounds During Dynamic BDD Minimization, *in Proc. 36th ACM/IEEE Design Automation Conference (New Orleans, LA)*, 1999, 29-32.

[DST+94] R. Drechsler, A. Sarabi, M. Theobaldi, B. Becker, M. A. Perkowski, Efficient Representation and Manipulation of Switching Functions based on Ordered Kronecker Functional Decision Diagrams, *in Proc. of the 31th IEEE/ACM Design Automation Conference*, 1994, 415-419.

[FFK88] M. Fujita, H. Fujisawa, N. Kawato, Evaluation and Improvements of Boolean Comparison Method based on Binary Decision Diagram, *in Proc. 25th ACM/IEEE Design Automation Conference*, 1988, 2-5.

[FHS78] S. Fortune, J. Hopcroft, E. Meineche Schmidt, The Complexity of Equivalence and Containment for Free Single Variable Program Schemes, *in Proc. of the 5th International Colloquium on Automata, Languages, and Programming (ICALP)*, LNCS **62**, Springer, 1978, 227-240.

[FMK91] M. Fujita, Y. Matsunaga, T. Kakuda, On Variable Ordering of Binary Decision Diagrams for the Application of Multi-Level Logic Synthesis, *in Proc. European Design Automation Conference (Amsterdam)*, 1991, 50-54.

[FOH93] H. Fujii, G. Ootomo, C. Hori, Interleaving Based Variable Ordering Methods for Ordered Binary Decision Diagrams, *in Proc. IEEE/ACM Int. Conf. on Computer Aided Design (ICCAD'93)*, 1993, 38-41.

[FS90] S. J. Friedman, K. J. Supowit, Finding the Optimal Variable Order for Binary Decision Diagrams, *in IEEE Transactions on Computers* **39**, 1990, 710-713.

[GM93] J. Gergov, Ch. Meinel, Frontiers of Feasible and Probabilistic Feasible Boolean Manipulation with Branching Programs, *in Proc. 10th Annual Symp. on Theoretical Aspects of Computer Science*, **665** of LNCS, Springer, 1993, 576-585.

[GM94a] J. Gergov, Ch. Meinel, Efficient Analysis and Manipulation of OBDDs can be Extended to FBDDs, *in IEEE Transactions on Computers*, **43(10)**, 1994, 1197-1209.

[GM94b] J. Gergov, Ch. Meinel, On the Complexity of Analysis and Manipulation of Boolean Functions in Terms of Decision Diagrams, *in Information Processing Letters* **50**, 1994, 317-322.

[GM96] J. Gergov, C. Meinel, Mod2-OBDDs: A Data Structure that generalizes EXOR-sum-of-products and Ordered Binary Decision Diagrams, *in Formal Methods in System Design* **8**, Kluwer Academic Publishers, 1996, 273-282.

[Gup92] A. Gupta, Formal Hardware Verification Methods: A Survey, *in Formal Methods in System Design*, vol.**1**, no.**4**, 1992, 335-383.

[HS96] G. D. Hachtel, F. Somenzi, Logic Synthesis and Verification Algorithms, *Kluwer Academic Publishers*, 1996.

[Hof95] D. R. Hof, Intel takes a Bullet - and Barely breakes Stride, *in Bussiness Week*, January 1995, 38-39.

[ISY91] N. Isiura, H. Sawada, S. Yajima, Minimization of Binaty Decision Diagrams based on the Exchange of Variables, *in Proc. IEEE International Conference on Computer Aided Design (Santa Clara, CA)*, 1991, 472-475.

[JAB+92] J. Jain, M. Abadir, J. Bitner, D. S. Fussell, J. A. Abraham, IBDDs: An Efficient Functional Representatikon for Digital Designs, *in Proc of the European Conference on Design Automation* (1992), 440-446.

[JBF+92] J. Jain, J. Bitner, D. S. Fussell, J. A. Abraham, Probabilistic Verification of Boolean Functions, *in Formal Methods in System Design* **1**, 1992, Kluwer Academic Publishers, 65-116.

[JPH+91] S.-W. Jeong, B. Plessier, G. Hachtel, F. Somenzi, Extended BDD's: Trading off Canonicity for Structure in Verification Algorithms, *in Proc. ACM/IEEE International Conference on Computer Aided Design*, 1991, 464-467.

[Kar88] K. Karplus, Representing Boolean Functions with If-Then-Else DAGs, *Technical Report UCSC-CRL-88-28, Computer Engeneering, University of California at Santa Cruz*, 1988.

[KSR92] U. Kebschull, E. Schubert, W. Rosenstiel, Multilevel Logic Synthesis based on Functional Decision Diagrams, *in Proc. of the European Design and Test Conference*, 1992, 43-47.

[KVB+98] T. Kam, T. Villa, R. Brayton, A. Sangiovanni-Vincentelli, Multi-Valued Decision Diagrams: Theory and Applications, *Multiple-Valued Logic, An International Journal* **4**, 1998, 9-62.

[Lee59] C. Y. Lee, Representation of Switching Functions by Binary Decision Programs, *in Bell System Technical Journal*, vol. **38**, 1959, 985-999.

[LGS] LGSynth93 Benchmarks:
`http://www.cbl.ncsu.edu/CBL_Docs/lgs91.html`.

[LN86] R. Lidl, H. Niederreiter, Introduction to Finite Fields and their Applications, *Cambridge University Press*, 1986.

[Lon93] D. Long, Model Checking, Abstraction and Compositional Verification, *Ph.D. Thesis, Carnegie Mellon University*, 1993.

[LPV94] Y.-T. Lai, M. Pedram, S. B. K. Vrudhula, EVBDD-based Algorithms for Integer Linear Programming, Spectral Transformation and Function Decomposition, *in IEEE Transactions on Computer Aided Design od Integrated Circuits and Systems* **13**, 1994, 959-975.

[LS92] Y.-T. Lai, S. Sastry, Edge Valued Binary Decision Diagrams for Multi-Level Hierarchical Verification, *in Proc. 29th ACM/IEEE Design Automation Conference*, 1992, 608-613.

[MB88] J.-C. Madre, J.-P. Billon, Proving Circuit Correctness using Formal Comparison between Expected and Extracted Behaviour, *in Proc. 25th ACM/IEEE Design Automation Conference (Anaheim, CA)*, 1988, 205-210.

[McC86] E. J. McClusky, Logic Design Principles, *Prentice-Hall*, 1986.

[Mei88] Ch. Meinel, Modified Branching Programs and their Computational Power, Habilitationsschrift, Humboldt-Universität Berlin, 1988. Published as *Lecture Notes in Computer Science LNCS* **370**, Springer, 1988.

[Mei90] Ch. Meinel, Polynomial Size Ω-Branching Programs and their Computational Power, *Information and Computation* **85**, 1990, 163-182.

[Mil93] D. M. Miller, Multiple-Valued Logic Design Tools, *Proc. 23rd Int. Symp. on MVL*, 1993, 2-11.

[MIY90] S. Minato, N. Ishiura, S. Yajima, Shared Binary Decision Diagrams with Attributed Edges, *in Proc. 27th ACM/IEEE Design Automation Conference (Florida, FL)*, 1990, 52-57.

[Moo65] G. E. Moore, Cramming more Components onto Integrated Circuits, *Electronic Magazine*, vol.**38**, no.**8**, 1965, 114-117.

[Moo95] G. E. Moore, Lithography and the Future of Moore's Law, *in Proc. of the SPIE*, vol.**2440**, 1995, 2-17.

[MR95] R. Motwani, P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

[MS97] C. Meinel, A. Slobodova, Speeding Up Variable Ordering of OBDDs, *in Proc. International Conference on Computer Design (Austin, TX)*, 1997, 338-343.

[MS97a] C. Meinel, H. Sack, Case Study: Manipulating $\oplus$-OBDDs by Means of Signatures, *in Proc. of the 3rd International Workshop on Applications of the Reed-Muller Expansion in Circuit Design (Oxford, UK)*, 1997, 175-186.

[MS98] C. Meinel, H. Sack, $\oplus$-OBDDs - A BDD Structure for Probabilistic Verification, *in Proc. Workshop on Probabilistic Methods in Verification*, 1998, 141-151.

[MS99] C. Meinel, H. Sack, Algorithmic Considerations of $\oplus$-OBDD Reordering, *in Proc. of the 4th International Workshop on Applications of the Reed-Muller Expansion in Circuit Design (Victoria, BC, Canada)*, 1999, 197-184.

[MS00] C. Meinel, H. Sack, Mod2OBDDs - a BDD Structure for Probabilistic Verification, *in Electronic Notes in Theoretical Computer Science*, vol.**22**, 2000.

[MT98] Ch. Meinel, T. Theobald, Algorithms and Data Structures in VLSI Design: OBDD - Foundations and Applications, *Springer*, Heidelberg, 1998.

[Mul54] D. E. Muller, Application of Boolean Algeba to Switching Circuit Design and Error Detection, *in IRE Transactions on Electronic Computing* **EC-3**, 1954, 6-12.

[MW86] J. C. Muzio, T. C. Wesselkamper, *Multiple-Valued Switching Theory*, Adam Hilger Ltd Bristol and Boston, 1986.

[MWB88] S. Malik, A. R. Wang, R. K. Brayton, A. Sangiovanni-Vincentelli, Logic Verification Using Binary Desicion Diagrams in a Logic Synthesis Environment, *in Proc. of the 25th ACM/IEEE Design Automation Conference*, 1988, 268-271.

[No+99] A. Nozoe et al., A 256-Mb Multilevel Flash Memory with 2 MB/s Program Rate for Mass Storage Applications, *in Proc. of 1999 IEEE Int. Solid-State Circuits Conference (ISSCC'99)*, 1999, 110-111.

[OM97] T. Okuda, T. Murotani, A Four-Level Storage 4-Gb DRAM *in IEEE Journal of Solid-State Circuits* **32**, 11, 1997, 1743 - 1747.

[PS95] S. Panda, F. Somenzi, Who are the Variables in your Neighbourhood, *in Proc. International Conference on Computer Aided Design (San Jose, CA)*, 1995, 74-77.

[Raz87] A. A. Razborov, Lower Bounds on the Size of Bounded Depth Networks over a Complete Basis with Logical Addition, *in Math. Notes* **41**(4), 1987, 333-338.

[Rud93] R. Rudell, Dynamic Variable Ordering for Ordered Binary Decision Diagrams, *in Proc. International Conference on Computer Aided Design (Santa Clara, CA)*, 1993, 42-47.

[Ree54] L. S. Reed, A Class of Multiple Error-Correcting Codes and their Decoding Scheme, *in IRE Transactions on Information Theory* **4**, 1954, 38-42.

[SDM00] H. Sack, E. Dubrova, Ch. Meinel, Representation of Multiple-Valued Functions with Mod-p-Decsion Diagrams, *in Proc. of IEEE/ACM Int. Workshop of Logic Synthesis (IWLS2000), (Dana Point, CA, USA)*, 2000, 341-348.

[Sch80] J. Schwartz, Fast Probabilistic Algorithms for Verification of Polynomial identities, *Journal of the ACM*, **27**, 1980, 701-717.

[Sha49] C. E. Shannon, The Synthesis of Two-Terminal Switching Circuits, *in Bell Systems Technical Journal* **28**, 1949, 59-98.

[SKM+90] A. Srinivasan, T. Kam, S. Malik, R. Brayton, Algorithms for Discrete Function Manipulation, *in Proc. International Conference on Computer Aided Design*, 1990, 92-95.

[Som96] F. Somenzi, CUDD: Colorado University Decision Diagram Pachage, *ftp://vlsi.colorado.edu/pub/*, 1996.

[SS98] R. S. Stankovic, T. Sasao, Decision Diagrams for Discrete Functions: Classification and Unified Interpretation, *in Proc. of Asia and South Pacific Design Automation Conference*, 1998, 439-446.

[SSL+92] E. M. Sentovitch, K. J. Singh, L. Lavagno, et al., SIS A System for Sequential Circuit Synthesis, *Technical Report UCB/ERL M92/41, Electronics Research Labs, University of California, Berkeley*, 1992. d

[STP86] P. A. Scott, S. E. Tavares, L. E. Peppard, A Fast VLSI Multiplier for $GF(2^m)$, *in IEEE Journal of Selected Areas of Communication* vol.**4**, 1986,62-66.

[SW93] D. Sieling, I. Wegener, Reduction of OBDDs in Linear Time, *in Information Processing Letters* **48**, 1993, 139-144.

[SW95] D. Sieling, I. Wegener, Graph Driven BDDs - A New Data Structure for Boolean Functions, *Theoretical Computer Science* **141**, 1995, 283-310.

[Thi00] T. Thierauf, The Computational Complexity of Equivalence and Isomorphism Problems, Lecture Notes in Computer Science **1852**, Springer-Verlag, 2000.

[VIS96] The VIS Group, VIS: A System for Verification and Synthesis, *in Proc. 8th Int. Conf. on Computer Aided Verification*, Springer Lecture Notes in Computer Science, **1102**, Edited by R. Alur and T. Henzinger, New Brunswick, NJ, 1996, 428-432.

[Waa97] S. Waack, On the Descriptive and Algorithmic Power of Parity Ordered Binary Decision Diagrams, *in Proc. 14th Symp. on Theoretical Aspects of Computer Science*, **1200** of LNCS, Springer, 1997.

[Weg87] I. Wegener, The Complexity of Boolean Functions, *Series in Computer Science, Wiley-Teubner, Stuttgart/Chichester*, 1987.

[Weg00] I. Wegener, Branching Programs and Binary Decision Diagrams : Theory and Applications, (*Siam Monographs on Discrete Mathematics and Applications*), *Society for Industrial & Applied Mathematics*, July 2000.

[Zip70] R. Zippel, Probabilistic Algorithms for Sparse Polynomials, *in Proc. International Symposium on Symbolic and Algebraic Computation*, Lecture Notes in Computer Science **72**, Springer-Verlag, 1979, 216-226.