
**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF INTELLIGENT SYSTEMS**
**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**


# EFFICIENT HANDLING OF BOOLEAN FUNCTIONS

**EFEKTIVNÍ PRÁCE S BOOLEOVSKÝMI FUNKCEMI**

**MASTER'S THESIS**
**DIPLOMOVÁ PRÁCE**


**AUTHOR** **Bc. JÁN MAŤUFKA**
**AUTOR PRÁCE**

**SUPERVISOR** **Ing. ONDŘEJ LENGÁL, Ph.D.**
**VEDOUCÍ PRÁCE**



**BRNO 2025**

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

# Master's Thesis Assignment

165012

| | |
|---|---|
| Institut: | Department of Intelligent Systems (DITS) |
| Student: | **Maťufka Ján, Bc.** |
| Programme: | Information Technology and Artificial Intelligence |
| Specialization: | Mathematical Methods |
| Title: | **Efficient Handling of Boolean Functions** |
| Category: | Theoretical Computer Science |
| Academic year: | 2024/25 |

Assignment:

1. Study the current state of the art in the area of compact representation of Boolean functions. Focus, in particular, on *binary decision diagrams (BDDs) [1]* and their extensions, such as *zero-suppressed decision diagrams (ZDDs) [2]*, *tagged BDDs (TBDDs) [3], chain-reduced BDDs* and *ZDDs (CBDDs/CZDDs) [4]*, and *BDDs with edge-specified reductions (ESRBDDs) [5]*.
2. Study the work [6], which introduces the ABDD data structure: BDDs parameterized with reductions and their canonic form.
3. Design an algorithm for efficient execution of Boolean operations over ABDDs.
4. Implement the designed algorithm and compare to algorithms in [1-5].
5. Discuss the achived results and propose extensions.

Literature:

[1] Randal E. Bryant: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers 35(8): 677-691 (1986)

[2] Shin-ichi Minato: Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. DAC 1993: 272-277

[3] Tom van Dijk, Robert Wille, Robert Meolic: Tagged BDDs: Combining reduction rules from different decision diagram types. FMCAD 2017: 108-115

[4] Randal E. Bryant: Chain Reduction for Binary and Zero-Suppressed Decision Diagrams. J. Autom. Reason. 64(7): 1361-1391 (2020)

[5] Junaid Babar, Chuan Jiang, Gianfranco Ciardo, Andrew S. Miner: Binary Decision Diagrams with Edge-Specified Reductions. TACAS (2) 2019: 303-318

[6] J. Maťufka. New Techniques for Compact Representation of Boolean Functions. BSc thesis. FIT BUT, Brno. 2023.

Requirements for the semestral defence:
First two items of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

| | |
|---|---|
| Supervisor: | **Lengál Ondřej, Ing., Ph.D.** |
| Head of Department: | Kočí Radek, Ing., Ph.D. |
| Beginning of work: | 1.11.2024 |
| Submission deadline: | 21.5.2025 |
| Approval date: | 31.10.2024 |

# Abstract

Binary decision diagrams (BDDs) are an important data structure in model checking, allowing efficient representation of Boolean functions. When working with BDDs, efficient manipulation of their structure is also often required. One of the most important functions on BDDs is Apply. Apply takes two BDDs representing Boolean functions, a Boolean operator, and produces a BDD representing their combined semantics. The main goal of this work is to develop an algorithmic framework for the Apply function operating on binary decision diagrams with reduction rules based on tree automata (ABDDs). The algorithm works such that no unnecessary unfolding of the reduced structure is done. The complexity of the algorithm is not dependent on the number of variables but on the initial graph sizes of the two input ABDDs.



# Abstrakt

Binárne rozhodovacie diagramy (BDD) sú dôležitou dátovou štruktúrou v model checkingu, umožňujúcou efektívnu reprezentáciu Booleovských funkcií. Pri práci s BDD sa často vyžaduje aj efektívna manipulácia s ich štruktúrou. Jednou z najdôležitejších operácií pre manipuláciu BDD je Apply. Apply vezme na vstup 2 BDD reprezentujúce Booleovské funkcie a Booleovský operátor, a vytvorí BDD reprezentujúci ich kombinovanú sémantiku. Hlavným cieľom tejto práce je vytvoriť algoritmický rámec pre funkciu Apply pracujúcu s binárnymi rozhodovacími diagramami s redukčnými pravidlami založenými na stromových automatoch (ABDD). Algoritmus funguje tak, aby nebolo potrebné nadbytočne rozbaľovať redukované štruktúry. Zložitosť algoritmu tým pádom nie je závislá na počte premenných, ale na veľkostiach grafov vstupných dvoch ABDD.



# Keywords

Binary decision diagram, Tree automaton, Apply.

# Kľúčové slová

Binárny rozhodovací diagram, Stromový automat, Apply.


# Reference

MAŤUFKA, Ján. *Efficient Handling of Boolean Functions.* Brno, 2025. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondřej Lengál, Ph.D.

# Rozšírený abstrakt

Binárne rozhodovacie diagramy (BDD) sú známou dátovou štruktúrou (orientovaný acyklický graf s koreňovým vrcholom) na efektívnu reprezentáciu Booleovských funkcií. Využívajú sa najmä vo formálnej verifikácii, model checkingu, návrhu digitálnych obvodov. V praktických aplikáciách sa často vyžaduje efektívna manipulácia s BDD, najmä funkcia Apply, ktorá skombinuje dva BDD reprezentujúce Booleovské funkcie $f, g$ pomocou Booleovského operátora $\odot$, a vytvorí BDD reprezentujúce funkciu $f \odot g$.

Funkcia Apply je algoritmus, ktorý rekurzívne prechádza oba vstupné BDD naraz, pričom "zarovnáva" ich štruktúry (umelým vkladaním vrcholov s konkrétnymi premennými, keďže je možné kombinovať len vrcholy s rovnakými premennými). Výsledný algoritmus má asymptotickú zložitosť, ktorá je súčinom veľkostí vstupných grafov, čo je dosiahnuté použitím techník ako memoizácia, dynamické programovanie, a skrátené vyhodnocovanie Booleovských výrazov.

Pre rôzne špecifické prípady sú BDD neefektívne, a preto za posledných 30 rokov vzniklo viacero modifikácií BDD, ktoré využívajú rôzne spôsoby redukcie. Jedným z najnovších prístupov v oblasti skúmania rôznych modifikácií BDD sú *binárne rozhodovacie diagramy založené na automatoch* (ABDD). ABDD používajú robustný prístup pre reprezentáciu redukčných pravidel, založený na stromových automatoch, ktorý umožňuje vytvárať vlastné redukčné pravidlá, prípadne meniť poradie, v akom sa budú uplatňovať. Každé pravidlo reprezentované stromovým automatom je potom schopné pokryť istý typ opakujúcich sa vzorov (podstromov v jazyku daného automatu).

Hoci existujú algoritmy, ktoré z obyčajného BDD získajú kanonickú podobu ABDD (kanonickosť v tomto kontexte znamená, že každá Booleovská funkcia je reprezentovaná práve jedným ABDD), zatiaľ nebol popísaný efektívny spôsob, ako nad ABDD vykonať Apply. Súčasné riešenie vyžaduje rozbalenie redukčných pravidiel do ich automatových štruktúr a následné rozvinutie slučiek, čo spôsobí exponenciálny nárast veľkosti štruktúry.

Cieľom tejto práce je implementovať efektívnejší spôsob vykonávania Apply nad ABDD, podobný ostatným BDD modelom. Zložitosť algoritmu nesmie byť závislá na počte premenných, ale iba na veľkostiach vstupov. Algoritmus musí byť schopný zlúčiť redukované hrany v ABDD (na základe daného operátora $\odot$), a to tak, aby nebolo nutné rozbaľovať ich sémantiku. Ďalej musí využívať rôzne optimalizačné techniky (memoizácia, dynamické programovanie, skoré vyhodnocovanie Booleovských výrazov), aby sa dosiahla žiadaná efektivita.

Kombinovanie redukčných pravidiel podľa Booleovského operátora sa dá vykonávať s použitím automatového produktu a špecifickým vytváraním koncových hrán na základe sémantiky operátora. Keďže stromových automatov reprezentujúcich redukcie je v ABDD konečný počet, je možné si dopredu predpočítať všetky produkty rôznych kombinácií automatov a operátorov, so všetkými potrebnými informáciami tak, aby počas algoritmu sa dalo okamžite zistiť, ako bude vyzerať skombinovaná hrana (teda redukčné pravidlo danej hrany a vrcholy, ktoré daná hrana spája).

Ďalším dôležitým faktorom je popísanie spôsobu, akým sa synchronizuje rekurzívny prechod algoritmu Apply. Inak povedané, ako redukované hrany ABDD možno rozdeliť umelým vložením vrchola s konkrétnou premennou tak, aby bolo možné synchronizovane pokračovať v algoritme ďalej. Tento proces *materializácie vrcholov* je navrhnutý tak, aby sa rozdelením hrany nenarušila jej sémantika. Podobne aj tu je len obmedzený počet spôsobov, akým bude vyzerať podštruktúra ABDD, ktorá vznikne rozseknutím redukovanej hrany za rôznych okolností. Preto je možné si všetky tieto situácie dopredu predpočítať a v prípade

potreby je možné ihneď vybrať správnu podštruktúru, ktorou sa upraví vstup, čím sa umožní synchronizovaný prechod oboma vstupnými grafmi.

Jednou nevýhodou algoritmu je, že jeho výsledok nemusí byť kanonický, čím sa vyžaduje následné použitie kanonizačných algoritmov. Momentálne nie je jasné, ako by bolo možné systematicky kontrolovať a redukovať novovzniknuté hrany a uzly, ak by nejakým spôsobom narušovali kanonicitu.

Vykonané experimenty potvrdzujú výsledky z pôvodnej práce, použili sa rovnaké testovacie vstupy, iba v rôznych štádiách konštrukcie (počas konštrukcie daných vstupov sa opakovane používa Apply). Toto simuluje bežné aplikácie, v ktorých sa s BDD opakovane manipuluje.

ABDD dosahujú v priemere 10-20 % menšie grafy ako iné modely. V teórii by to malo znamenať, že čím menšie grafy sa dajú reprezentovať pomocou ABDD, tým efektívnejší by mal byť Apply v porovnaní s inými modelmi, keďže asymptotická zložitosť Apply je lineárne závislá od veľkosti vstupov.

Hľadaním iných opakujúcich sa vzorov a pridávaním nových redukčných pravidiel sa efektivita ABDD reprezentácie, a tým pádom aj efektivita Apply dá zvyšovať, čo poukazuje na potenciál tohto prístupu.

# Efficient Handling of Boolean Functions

## Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Ondřej Lengál, Ph.D. and that all literary sources, publications, and other sources were properly listed and cited.

.....
Ján Maťufka
May 21, 2025


## Acknowledgements

I would like to thank my supervisor Ing. Ondřej Lengál, Ph.D. for his guidance, patience, help and ideas while working on this project. I am deeply grateful to my parents for their endless love, patience and support. Without their sacrifices and encouragement, this work would not have been possible. I hope I can use the knowledge and experience gained during this academic journey to make them proud.

# Contents

# Chapter 1

# Introduction

Binary decision diagrams (BDDs) are extensively used in computer science as an efficient way to represent Boolean functions. They have a wide variety of uses in formal verification, model checking, computer-aided design (CAD), and other areas of mathematics and computer science. For practical purposes, efficient manipulation and canonicity are required by any model that tries to represent Boolean functions. Canonicity for BDDs means that there only exists one BDD that can represent a given Boolean function. This is often enforced by specifying an algorithm that tries to reduce the BDD structure to its minimal form.

However, enforcing canonicity becomes a challenge when trying to efficiently implement other operations that manipulate BDD structures. One such operation, Apply, takes two BDDs representing Boolean functions $f, g$, a binary logical operator $\odot$, and returns a BDD representing $f \odot g$. Apply is optimized so that it does not have to expand the reduced structure if not necessary.

A big area of BDD research focuses on utilizing special reduction rules to reduce the memory complexity of the graph representations. One of the recent attempts [24] uses reductions that label edges of the BDD graph, each reduction representing a tree automaton that accepts trees that represent some reducible pattern. This model is called Automata-based binary decision diagrams (ABDDs). ABDDs suffer from one limitation and that is that the inefficiency of Apply. The initial proposal required that the reduced structure is unfolded, the loops left in the unfolded structure unwound (so essentially turning the decision diagram into a binary tree), and after that the operator is performed, which then allows to reduce the resulting structure.

The goal of this work is to implement a more efficient Apply algorithm on ABDDs that does not require unwinding of the loops. Instead, the algorithm should utilize reduction rule merging and all possible optimization techniques (memoization, early evaluation, etc.) to perform this more efficiently. The complexity of the algorithm should not be dependent on the number of variables (as is the case with the loop unwinding version), but on the graph sizes of the two input operands.

Binary decision diagrams (BDDs) are introduced in Chapter 2, along with their modifications relevant to this work. In Chapter 3, the Apply algorithm is explained, its implementation, optimization techniques, and its modifications in other BDD models. All relevant theoretical foundations of ABDDs, tree automata, and a description of ABDD canonization are given in Chapter 4. Chapter 5 focuses on the implementation of Apply on ABDDs (and the intuition behind it), all techniques used to make the algorithm work efficiently within the expected bounds. Finally, Chapter 6 overviews the experimental results and highlights the strengths and weaknesses of ABDDs.

# Chapter 2

# Binary Decision Diagrams

This chapter serves as an introduction to the concept of binary decision diagrams [9] (BDDs, singular form: BDD), how they look like, how they work, and how they relate to Boolean functions. Along with how to obtain a canonic BDD representation of a Boolean function, the chapter focuses on practical applications of BDDs, while mentioning challenges, limitations, and other areas of interest concerning BDDs.

The main portion of this chapter introduces the reduction techniques employed in the state-of-the-art models that modify or expand upon the standard BDDs in order to reduce the spatial complexity (or improve some other aspect of BDDs) – these models include Zero-suppressed BDDs [25], Tagged BDDs [34], Chain-reduced BDDs [8], BDDs with edge-specified reductions [4] (and their counterpart utilizing complemented edges [3]).

Understanding the inner workings and reductions of BDD-based representations is key to understanding the motivation behind the goal of this thesis and the main idea behind the approach used to achieve it.

## 2.1 Boolean functions

A key concept in discrete mathematics, Boolean functions have been a major subject of study since the middle of 19th century, when George Boole (after whom they are named) laid the foundations of mathematics in logic.

A Boolean function is any function whose arguments, as well as the function itself, assume values from a two-element set (usually $\{0, 1\}$). Formally speaking, a Boolean function $f$ has the following signature: $f : \{0, 1\}^n \to \{0, 1\}$, where $n \in \mathbb{N}$ [7]. There are multiple ways to represent (specify/define) a Boolean function (see Figure 2.1):

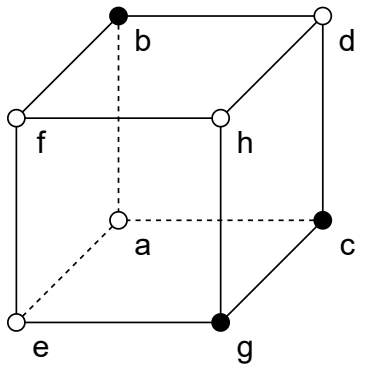
- Truth tables – systematically writing down every possible assignment of truth values to variables along with the resulting truth value of the specific assignment in a table format. Note that the number of rows of the truth table grows exponentially with the number of variables.
- Propositional logic formulae/expressions – assigning truth values to variables and evaluating the expression based on the semantics of the logical connectives (the logical connectives themselves are simple unary/binary Boolean functions).
- Combinational circuits – basically a graphical way to represent a propositional formulae, where input signals correspond to truth values assigned to variables and simple logic gates correspond to logical connectives (in this context, instead of talking about

true/false values, low/high is usually used since it more closely relates to the voltages going through the circuit).

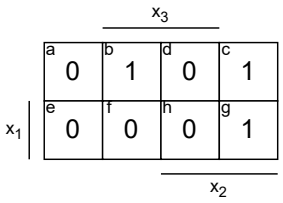
- Karnaugh maps – a grid-like visual way of representing the truth table of Boolean functions (usually best suited for functions of two to four variables), such that variable assignments that differ in one variable's truth value are next to each other in the grid (Gray's code). Karnaugh maps (or K-maps) are used to simplify Boolean expressions while designing digital circuits (so that they consist of less logic gates).
- $N$-dimensional cubes – every vertex of the $N$-dimensional cube represents one variable assignment. Similarly to K-maps, vertices representing "neighboring" assignments share an edge on the $N$-dimensional cube. In a similar fashion, cube's planes of higher dimensions will have less variables set to the same value. Vertices of a $N-1$ dimensional plane of the cube will have exactly one variable set to a specific truth value. [7]

| | $x_1$ | $x_2$ | $x_3$ | $f_1$ |
|---|---|---|---|---|
| a: | 0 | 0 | 0 | 0 |
| b: | 0 | 0 | 1 | 1 |
| c: | 0 | 1 | 0 | 1 |
| d: | 0 | 1 | 1 | 0 |
| e: | 1 | 0 | 0 | 0 |
| f: | 1 | 0 | 1 | 0 |
| g: | 1 | 1 | 1 | 0 |
| h: | 1 | 1 | 1 | 0 |

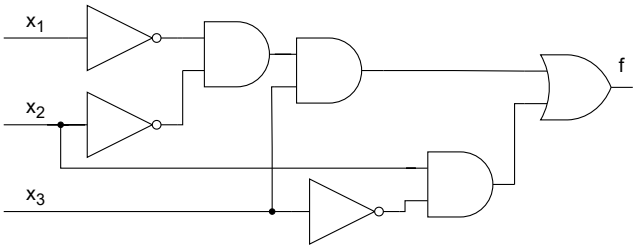
(a) Truth table of Boolean function $f_1$




(b) $f_1$ as an N-dimensional cube




(c) $f_1$ as a Karnaugh map




(d) Combinational circuit representing $f_1$

Figure 2.1: Boolean function $f_1(x_1, x_2, x_3) = (x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge x_3)$ represented in multiple ways. Note how the adjacent vertices of the N-dimensional cube and adjacent cells of a Karnaugh map (even adjacent through wrap-around) differ in just one variable value. This demonstrates the principle of *Boolean adjacency* – Boolean terms (assignment of truth values to variables) are adjacent if they differ in exactly one value. Also notice how the number of propositional connectives is the same as the number of logic gates in the combinational circuit representation.

An important thing to note with regards to Boolean functions is that for any Boolean function with $n$ inputs, there are $2^n$ different assignments, each of which can lead to one of two different truth values. Thus, for $n$ Boolean variables, there are $2^{2^n}$ different Boolean functions. So there are four unary Boolean functions (always 0, always 1, identity, negation), 16 binary Boolean functions, etc.

## Propositional logic

Boolean functions provide a mathematical framework for evaluating the truth values of propositional logic formulae (or expressions). The syntax of propositional logic consists of the following:

- The set of variables (which, upon evaluation, can only obtain the truth values 0 and 1).
- The truth values 0 and 1 (can be considered nullary functions – constants).
- The logical connectives:
  - Logical conjunction $\wedge$ – alternatively: AND, &, $\cdot$.
  - Logical disjunction $\vee$ – alternatively: OR, |, +.
  - Logical negation $\neg$ – alternatively NOT, $-$ or !, also represented by a line over an expression, like $\overline{x+y}$. While the other mentioned connectives are binary Boolean functions, logical negation is one of four unary Boolean functions (along with logical true, logical false, logical identity).
  - Logical implication $\rightarrow$ (conditional) and logical equivalence $\leftrightarrow$ (bi-conditional).
  - Other logical connectives, like exclusive or $\veebar$ (XOR, $\oplus$), Sheffer stroke $\uparrow$ (NAND), Pierce arrow $\downarrow$ (NOR).
- The parentheses are used when trying to enforce a priority order between connectives – by default, the logical connective precedence is the following: negation > conjunction > disjunction > conditional > bi-conditional.

A set of logical connectives is *functionally complete*, when it is able to express all possible truth tables by combining members of the set into a Boolean expression. Notable functionally complete sets are $\{\vee, \neg\}$ and $\{\wedge, \neg\}$ (they can be interchanged thanks to De Morgan laws), singleton sets $\{\uparrow\}$ and $\{\downarrow\}$. In practice, this means that if a set of connectives is able to express negation and either of conjunction/disjunction, then by using a combination of the connectives from the set, one can represent any Boolean function.

The most utilized functionally complete set of logical connectives in practice is $\{\neg, \vee, \wedge\}$. This set of connectives is used in all *normal forms* of propositional formulae. A propositional logic formula is in *negation normal form* (NNF) if, firstly, the only connectives used are $\neg, \vee, \wedge$ (and nullary functions $0, 1$), and secondly, negation connective is only used right in front of variables (i.e. in the innermost positions). A *literal* is defined as a Boolean variable or the negation of a Boolean variable. Negation normal form, can be alternatively defined as conjunction and disjunction of literals.

*Disjunctive normal form*, also called sum of products, OR of ANDs is defined by this structure:

$$\bigvee_i \bigwedge_j L_{ij} \text{ where } L_{ij} \text{ is a literal, i.e. either a variable or its negation.}$$

So, similarly to NNF, the only allowed connectives are $\neg, \vee, \wedge$, and negation is only allowed directly in front of variables. But, additionally, the whole formula is one disjunction of *clauses*, where clauses are defined as conjunctions of literals.

The last important normal form is the *conjunctive normal form* (CNF), also called a product of sums, AND of ORs. The propositional logic formula is in CNF when it adheres to the following structure:

$$\bigwedge_i \bigvee_j L_{ij} \text{ where } L_{ij} \text{ is a literal.}$$

Essentially, a formula in CNF is a conjunction of clauses, each clause is a disjunction of literals.

Most practical applications (SAT solvers, automated theorem provers) work with formulae in CNF because of the underlying heuristics of these tools. Search strategies, backtracking, efficient conflict detection, propagation, invalid branch pruning were all designed with the CNF structure in mind [15]. It is also more natural to represent constraints of the analyzed system as a list of what has to hold at all times. DNF focuses on enumerating solutions, which, even though is more natural for human understanding, leads to exponential growth and impracticality in most scenarios.

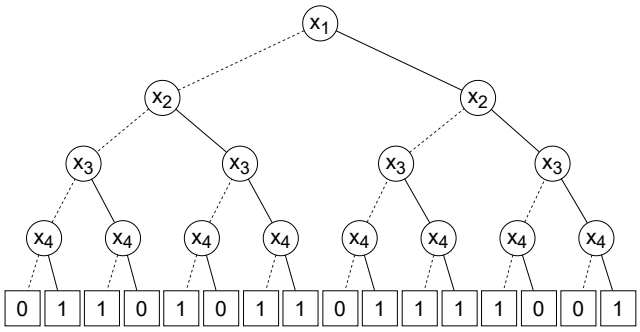
## 2.2 Representing Boolean functions as decision trees

On top of the earlier mentioned ways to specify or represent Boolean functions, there is also another, very intuitive way – decision trees. Specifically, in the case of Boolean functions, binary decision trees. Decision trees are used in many areas of computer science, like statistics, especially machine learning – practical algorithms such as ID3, C4.5, and CART, state space search in AI development, etc. [19]

Given a truth table, binary decision trees directly map to the way machines would think about evaluating the result of a Boolean function (given the specific assignment of truth values to each variable/argument of the function). Top-down decision tree traversal reflects a series of questions about the truth values of variables, upon arriving at the final result – a truth value in some leaf node. The non-leaf nodes of the tree select which branch to send the decision process to, based on the truth value of a particular variable represented by that node. Obtaining the binary decision tree, given a truth table of a Boolean function is depicted in Figure 2.2.

| $x_1x_2$ \ $x_3x_4$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 1 | 0 | 1 |
| 01 | 1 | 0 | 1 | 1 |
| 11 | 1 | 0 | 1 | 0 |
| 10 | 0 | 1 | 1 | 1 |

(a) A K-map of function $f_2$




(b) Full binary decision tree of function $f_2$

Figure 2.2: Boolean function represented as a decision tree. Dashed lines leading from nodes down represent the truth value of the variable in the node being false. Solid lines represent true.

Naturally, looking at some binary decision tree representing a Boolean function, it is clear that at some point during the traversal, the function result is already determined,

regardless of which branches are chosen until leaf nodes are reached or that the branch choice in some node is irrelevant. In the same manner, the decision tree can include nodes that are identical, i.e., the decision process (or the rest of it) will proceed in identical way upon reaching these nodes. These two redundancies are the main idea on which binary decision diagrams (BDDs) are built and how they are able to compactly represent (without any loss of semantics) Boolean functions.

The first notion of using decision diagrams as a means of effective representations of Boolean functions came in 1959 by Lee [21]. Back then, they were called binary-decision programs and Lee demonstrated their applications in designing switching circuits. The main takeaway was that they are more efficient representations of Boolean functions, while being harder to manipulate, as opposed to the algebraic forms of representation. Later, in 1978, Akers [2] introduced the current notion of binary decision diagrams. However, no formal basis or implementation details of BDDs were given. The paper only demonstrated methods for deriving the diagrams and gave a few examples for some basic combinational and sequential devices/circuits (including, for example, a *k-carry look-ahead adder*). It was Bryant's work in 1986 [9] that gave BDDs the structure and algorithmic framework it is known for today. What follows is a deeper look into the key aspects of how binary decision diagrams work.

## 2.3 Variable ordering matters

Very important aspect of BDDs that has not yet been mentioned here is the fact that one Boolean function can be represented by multiple BDDs. In particular, choosing a different order of variable evaluation leads to a different (fully expanded) binary decision tree – and as such, upon applying reductions, a different binary decision tree. A poor variable order can result in a significantly larger BDD, see Figure 2.3.

Choosing non-optimal variable order can significantly increase the resulting size of the BDD, ranging from linear to exponential growth. Based on the number of variables $n$, the upper bound for the number of nodes of the BDD is $(2^n/n)(2+\varepsilon))$, where $\varepsilon$ is an arbitrarily small positive number. Even though there is an exponential dependency, most functions are not that sensitive to variable ordering, i.e. the sizes of BDDs with the best and the worst ordering differ by a factor converging to 1 exponentially fast. [22, 37]

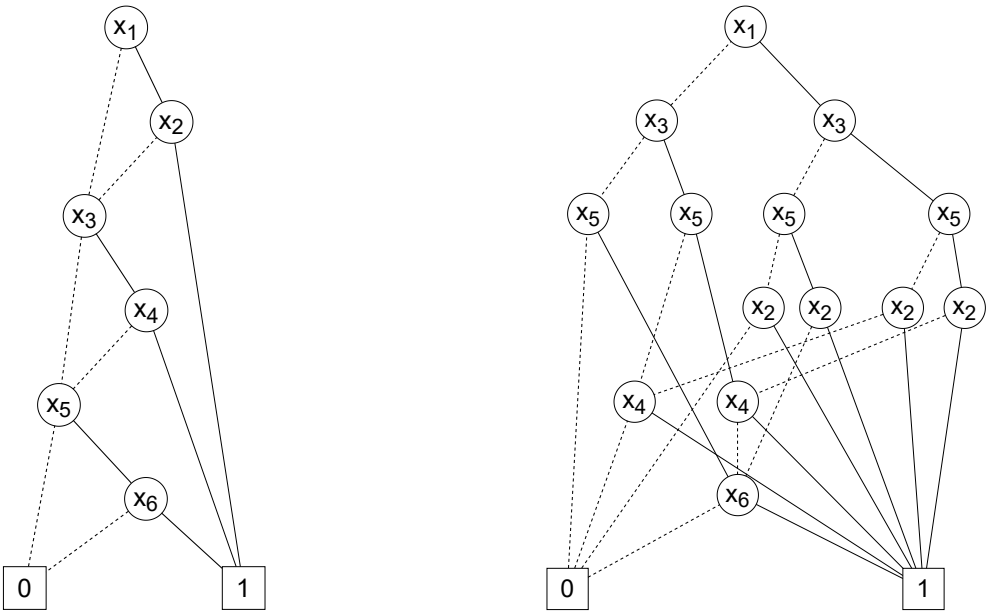
Obtaining an optimal variable order for a given Boolean function, such that the resulting BDD size is minimal, is NP-hard, even when the function is already represented as an ordered BDD [6].

Static reordering hopes to find the optimal variable order before the BDD is constructed. There are published algorithms that find the optimal variable ordering for a general Boolean function with $n$ variables, with the worst-case time complexity of $O(n^2 3^n)$ [17]. More practically, however, there are heuristic methods with much better time complexity deriving a BDD from specific Boolean function representation like circuits, CNF, or state machines [10].

Dynamic reordering tries to minimize the size complexity on an already constructed BDD. Introduced by Rudell [29], pairwise swapping of adjacent variables, while preserving external references to the graph's root vertices, thus not changing the semantics of the underlying function, achieves a more compact (not necessarily optimal) BDD structure. Sifting is another dynamic technique, and works by moving variables across potential positions in the order. When trying to swap positions of variables $x_i$ and $x_{i+1}$ in the variable ordering, for any occurrence of nodes with $x_i$ variable, sifting just swaps the nodes reached through

assigning $x_i = 1 \wedge x_{i+1} = 0$ and through $x_i = 0 \wedge x_{i+1} = 1$. Sifting is more computationally expensive, but it contributes massively to improving memory performance.

Overall, dynamic reordering is essential in real-world applications, where variable interdependencies and application-specific constraints sometimes make static optimization impossible. Achieving the best possible variable order while staying computationally feasible still remains an important area of BDD research.




(a) Optimal variable order $x_1 < x_2 < x_3 < x_4 < x_5 < x_6$

(b) Non-optimal variable order $x_1 < x_3 < x_5 < x_2 < x_4 < x_6$

Figure 2.3: Comparison of (canonical) BDDs obtained from the same function $f_3 = (x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6)$, but using a different variable order. Since the variable order is fixed, there is no need to use arrows, because the direction of edges can be inferred, from node $v$ to node $v'$, such that $var(v) < var(v')$. Figure adapted from [9].

## 2.4 Reducing the spatial complexity

Formally, BDD over an ordered set of variables $\mathbb{X} = \{x_1, \ldots, x_n\}$ is a directed acyclic graph (DAG) $G = (V, E)$, with one root vertex $r_v \in V$ (with no incoming edges, i.e. $\nexists v \in V : (v, r_v) \in E$) and two leaf vertices (no outgoing edges) – terminals $\mathbf{0} \in V$ and $\mathbf{1} \in V$ (denoted in bold). Vertices are sometimes referred to as nodes and terminal vertices are referred to as terminal nodes or leaf nodes. The terms non-terminal/internal/non-leaf are also used interchangeably. Every node of the BDD represents a separate Boolean function, evaluation of which starts at the root node. Terminal nodes represent trivial (nullary) Boolean functions 0 and 1. There are three functions associated with each non-leaf node of a BDD (i.e., the domain of the function is the set of nodes of the BDD):

- $var(v) : V \to \mathbb{X}$ returns a variable associated with it.
- $low(v) : V \to V \cup \{\bot\}$ represents the target node of the "low" outgoing edge of a node $v$. $low(v)$ returns a BDD that is reached when $v$ is evaluated is false. Alternatively, it can be thought of as a subgraph rooted in the low child node of $v$.

- $high(v) : V \to V \cup \{\bot\}$ is similar to function $low$, but for the "high" outgoing edge (evaluating $v$ as true). Note that $low$ and $high$ return $\bot$ for leaf nodes.

Considering that BDDs assume a fixed variable order, the following must hold for any non-leaf node $v$:

- if $low(v)$ is not a leaf, then $var(v) < var(low(v))$,
- if $high(v)$ is not a leaf, then $var(v) < var(high(u))$.

Essentially, $var$ gives each node an index that refers to the position of the variable (which the node's decision is based upon) in the given order. The restriction above means, that there is no backtracking or repeating in the variable order on all root-leaf paths of the BDD. It does not necessarily mean that every variable must appear on a given path. This allows BDD structure to sometimes "skip" variables in cases where the resulting truth value is not affected by that particular variable (in a given branch of the BDD). For nodes representing terminals **0** and **1** (leaf nodes), $var()$ function is not defined. Instead, $val(v)$ can be used to get the truth value stored within that node ($low$ and $high$ functions in this case also have no use).

BDDs with a fixed variable order are usually referred to as ordered binary decision diagrams (OBDDs). Ordered binary decision diagrams in a canonical form (obtained by applying reduction rules on the structure of the BDD until no more reduction is possible) are usually referred to as reduced ordered binary decision diagrams (ROBDDs). Note that canonicity inherently requires a fixed variable order. Canonical form is obtained by applying reduction rules in a systematic way, until no rules are applicable. Generally, by using systematic rule application and placing some structural restriction (patterns that are not allowed), canonical form makes it so that for a given Boolean function, there is exactly one possible ROBDD representation. For brevity, in this thesis, BDDs will refer to ROBDDs.

The biggest improvement of the BDDs is their compactness, which is achieved by these reduction principles:

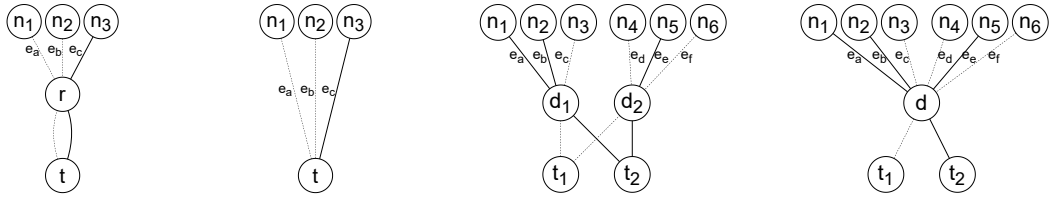
1. *Merging terminal nodes.* There is at most one leaf node with a given terminal value $(\mathbf{0}, \mathbf{1})$.
2. *Removing redundant nodes.* There is no node $v$ such that $low(v) = high(v)$, i.e. any node, whose low and high descendant are the same, is redundant.
3. *Merging duplicate nodes.* There cannot be distinct non leaf nodes $u, v$ such that $var(u) = var(v)$, $high(u) = high(v)$ and $low(u) = low(v)$, i.e., any isomorphic subgraphs of a BDD should be merged together.

In the case of the third rule, it should be noted that *isomorphic* subgraphs refer to such graphs $G$ and $G'$ that there exists a one-to-one function $f$ between the nodes of $G$ and nodes of $G'$ such that for any $f(v) = v'$ it holds that either:

- both $v$ and $v'$ are terminal nodes with the same value ($val(v) = val(v')$),
- or they are both non-terminal (non-leaf) nodes and $var(v) = var(v') \wedge low(v) = low(v') \wedge high(v) = high(v')$.

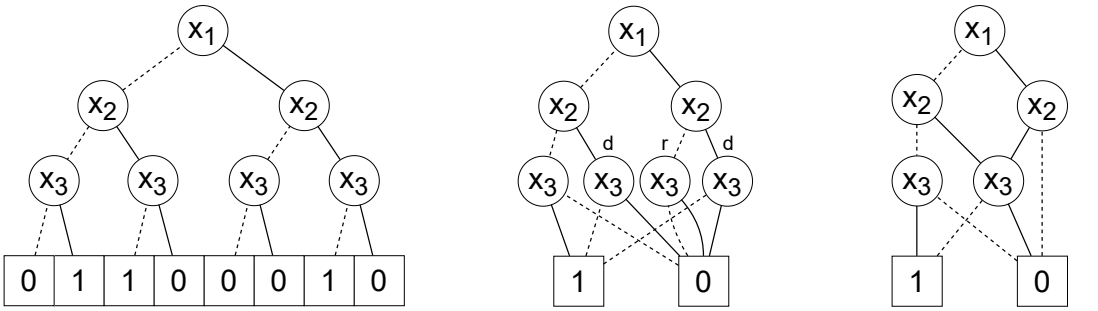
In a canonic form of a BDD, every node represents a different function. Duplicate nodes, or isomorphic subgraphs, represent an identical Boolean function, thus invalidating this property. Examples of the reducible patterns in a BDD structure are shown in Figure 2.4. For general (or random) Boolean functions, merging duplicate nodes reduces the size of BDDs better than removing redundant nodes [22].

Obtaining a reduced form of a BDD, which is also a canonic representation of a Boolean formula, given the particular variable ordering, works by first merging the terminal nodes, and then continually finding redundant and duplicate nodes and updating the BDD structure so that the semantics are not lost. For demonstration of the forbidden patterns and how the reduction process works, see Figure 2.5, for demonstration of how to take care of multiple references, see Figure 2.4.




(a) Redundant node (b) After removal (c) Two duplicate nodes (d) Merged duplicate nodes

Figure 2.4: Demonstration of forbidden (reducible) patterns in a ROBDD. Nodes labeled $t$ or $t_i$ already represent reduced BDD subgraphs, nodes labeled $n_i$ can be subject to further reductions based on the reduction algorithm's bottom-up approach. Edges are (for demonstration) labeled with $e$. Nodes labeled with $d$ have to have the same variable.




(a) Full binary decision tree (b) After merging leaves (c) After reductions

Figure 2.5: **BDD reduction process.** Boolean function $f_1$ from Figure 2.1, represented as a full binary decision tree (left), then after merging leaf nodes (middle), and after applying BDD reduction rules (right). In the middle picture, the redundant node is labeled $r$, and the duplicate nodes (same variable and same low and high child nodes) are labeled $d$.

In case of removing redundant nodes (low and high edges lead to the same node), all edges pointing to the redundant node need to be redirected to the descendant. In case of merging duplicate nodes (which represent the same function), edges leading to the node that will be removed also need to be redirected to its copy, preserving the integrity of the function. Merging duplicate nodes (and removing redundant ones) works by inductively building a mapping $id : V \to \mathbb{N}$ (where $V$ is a set of nodes of the BDD) of each unique node to an integer (unique integers for nodes in which unique subgraphs are rooted) in a bottom-up approach.

If for a currently processed node $v$ it holds that $id(low(v)) = id(high(v))$, then $v$ is redundant and $id(v) = id(low(v))$ should be set. If there is some other node $v'$ with the same variable as $v$ such that $id(low(v)) = id(low(v')) \wedge id(high(v)) = id(high(v'))$, then the subgraphs rooted at $v, v'$ are isomorphic and $id(v) = id(v')$ should be set.

## 2.5 Challenges with efficient implementations of BDDs

Symbolic model checking using BDDs has been a significant success in the area of advanced system verification, however, scaling to larger systems is still a problem. Even though modern computer architectures offer more memory, cores, and computing power, most BDD libraries do not utilize these resources efficiently because their implementations run on single-core [10]. The biggest bottleneck in efficient scaling is memory performance, since depth-first traversal and standard hashing in BDD library implementations results in poor memory locality, while improved locality implementations using breadth-first traversal and memory packing are not yet widely used. Issues with potential parallelism in BDD operations, like managing graph partitioning and communication overhead among nodes, seem to have been efficiently dealt with in the queue-based system that distributes work using tasks from the University of Twente [36] (which resulted in a parallel multi-terminal BDD library Sylvan [35]).

BDDs outperform SAT solvers in tasks that require exhaustive solution representations (counting solutions or variable quantification). In most applications, where only checking whether or not a Boolean function is satisfiable, SAT solver top-down techniques (DPLL algorithm, CDCL) perform better. Combining SAT solvers and BDDs could use the benefits of both approaches for specific problem types. [10]

### Representing non-Boolean domains using DDs

In many applications, decision diagrams can be utilized to represent functions with non-Boolean domains (. In the following, assume the domain over which functions are evaluated is $K$ (and the size of this domain is $\text{card}(K)$). If the underlying decision diagram is a BDD, the discrete domain will require a special encoding:

- **Binary encoding** maps variables to Boolean combinations (bit-vectors) using a minimal number of binary variables – this is the most memory efficient when the domain size is a power of two (DDs variable domain size is $n \cdot \log_2(\text{card}(K))$).
- **Unary encoding** uses one-hot encoding for variable values, practical for BDD model that works well with sparse functions (see Section 2.6) but inefficient in classic BDDs – this approach is less memory efficient, the larger the domain is (DDs variable count is $n \cdot \text{card}(K)$).

The other approaches that do not yield a BDD when representing generally discrete domains are:

- Multi-terminal binary decision diagrams (MTBDDs) represent functions with non-Boolean ranges, supporting algebraic operations. However, their size can grow exponentially due to large function images, limiting efficiency unless modularity in the system reduces this growth.

- Multiway branching ($n$-ary DDs) allows nodes to choose one of $n$ branches for its $n$-valued variable.

Decision diagrams can be also utilized to represent functions over unbounded domains. Of course, one can use discretization of the continuous/unbounded domain and utilize the above mentioned methods, but there are also special decision diagrams that can be used instead.

The most popular representatives of unbounded domain decision diagram (in this case, infinite domain) are difference decision diagrams (DDDs) [28]. Nodes in DDDs represent predicates (e.g., $x_i - x_j \leq c$), rather than Boolean variables. DDDs are heavily used in model checking of timed automata. DDDs have to represent constraints on real-valued clocks and must handle infeasible paths and redundant terms, which often requires exponential time for path elimination.

The capabilities of DDDs can be expanded using variants like clock difference diagrams [20], which are used in verification of timed automata, and analyzing real-time systems (merging or combining related constraints to reduce redundancy and improve efficiency) and linear decision diagrams [11] (designed to handle arbitrary linear constraints).

While mirroring SMT solvers by symbolically representing constraints across mathematical theories, these approaches still have troubles managing conflicts (or infeasible paths, or contradictions).

## 2.6 Zero-suppressed binary decision diagrams

*Zero-suppressed binary decision diagrams* (ZBDDs) were one of the first modifications of BDDs. First introduced by Minato in 1993 [25], they work exactly like BDDs, except they differ in the reduction rule. The motivation for ZBDDs came from the fact that in many applications, BDDs do not simply represent Boolean functions, but rather sets. Suppose a set of $n$ objects, from which combinations are selected. Each combination can be represented as a binary vector $(a_1, a_2, \ldots, a_n)$, where $a_i = 1$ if the $i$-th object belongs to the set, $a_i = 0$ otherwise. Then a combination set is the set of these binary vectors. Combination sets can be also understood as subsets of the power set of $n$ objects. A combination set $S$ of binary vectors of length $n$ corresponds to a Boolean function $f$ of $n$ variables, such that: $f(x_1, \ldots, x_n) = 1 \iff (x_1, \ldots, x_n) \in S$.

Such a Boolean function is called a *characteristic function* of the set of combinations. The output value of the characteristic means whether a combination specified by the input variables is included in the set or not. Solving combinatorial problems can be done through manipulating combination sets using operations like union, intersection, and difference. These set operations can be executed by logic operations on characteristic functions. When manipulating sets in combinatorial problems, Minato noticed that BDDs are not so efficient.

All paths in the BDD from the root node to terminal **1** represent all combinations in the set. When using BDDs to represent characteristic functions, combination set manipulation can be done quite efficiently. However, BDDs depend on the number of input variables, therefore the number of inputs has to be fixed before the BDD creation. In combination sets, variables representing irrelevant items not appearing in any combination (default variables) are regarded as zero when the characteristic function is true. Such variables cannot be suppressed in BDDs and many useless nodes for irrelevant objects have to be introduced. Node elimination is inefficient in dealing with such nodes. And so, ZBDDs were introduced.

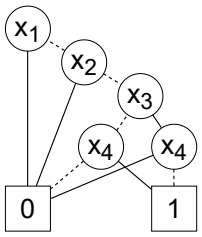
In BDDs, for a redundant node $n$ it holds that $low(n) = high(n)$, the reduction rule for redundant nodes in ZBDDs is changed to the following:

- Node $v$ should be eliminated when its high edge points to **0** (i.e. $high(v) = \mathbf{0}$). Edges reaching the eliminated node $v$ are redirected to $low(v)$

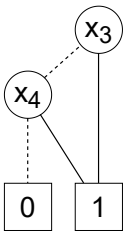
The rule about merging duplicate nodes (i.e. roots of isomorphic subtrees) still holds. The rule specific for ZBDDs is assymetric, so there is no alternative for low-edge pointing to **0**. For a demonstration of when ZBDD representation of a Boolean function is more effective than a BDD representation, see Figure 2.6.

| $x_1x_2$ \ $x_3x_4$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 1 | 0 | 1 |
| 01 | 0 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 |

(a) $f_4$ as a Karnaugh map




(b) $f_4$ as a BDD




(c) $f_4$ as a ZBDD

Figure 2.6: Comparison of BDD and ZBDD representation for a characteristic function $f_4(x_1, x_2, x_3, x_4) = (\neg x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4) \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4)$. Note the sparsity of the function $f_4$. The combination set represented by $f_4$ is $S = \{(0010), (0001)\}$. Alternatively $S$ can be written as $S = \{x_3, x_4\}$, where combination $c$ is written as a multiplication of items included in $c$. Figures are adapted from [25].

For ZBDD representations to be effective, the Boolean functions have to be sufficiently sparse. An experiment was conducted in [25], where the characteristic function represented a combination set of 100 combinations, each of which contained exactly $k$ items from 100. When $k$ was smaller, the ZBDDs were much more compact than BDDs. The effect was weakening with the increasing $k$. Another advantage of ZBDDs is the fact that the number of nodes from the root to the terminal node **1** is equal to the number of combinations in the combination set.

So, ZBDDs are more effective than BDDs, when the following criteria are met:

1. There are many input variables.
2. Default variables are regarded as zero.
3. The function represents sets of "sparse" combinations.
4. The number of combinations is large (otherwise, storing combinations in a linked list is better). [26]

Note that sets of sparse combinations are not the same as sparse sets of combinations. Sparse combinations are bit-vectors with a low number of ones. Sparse sets have a low number of vectors, which goes against the fourth criterion mentioned above. A good example when to use a ZBDD is storing solutions to $N$-queens problems. Since there are $N^2$ cells and the combinations will have only $N$ elements set to true, the bigger the $N$, the more efficient the representation (compared to BDDs).

ZBDDs have found applications in many areas. For example, in data mining, ZBDDs can be efficiently utilized in algorithms for mining frequent itemsets [27]. Another example is in graph optimization problems [13], such as:

- Maximal cliques. Find all cliques of a graph that are not a proper subset of another clique (finding a maximum cardinality clique is NP complete, finding all proper maximum cliques is exponential).
- Minimum $\alpha$-covering. Let $X, Y$ be two sets and $R \subseteq X \times Y . Y' \subseteq Y$ covers $X' \subseteq X$ when $\forall x \in X' \exists y \in Y' : xRy$. The problem is defined as finding a minimum cardinality set $Y' \subseteq Y$ that covers at least a fraction $\alpha$ of $X$, given the triple $(X, Y, R)$.
- Maximum $k$-cover. Find $k$ elements of $Y$ that cover as many elements of $X$ as possible, given the triple $(X, Y, R)$ (defined as above).

## 2.7 Tagged binary decision diagrams

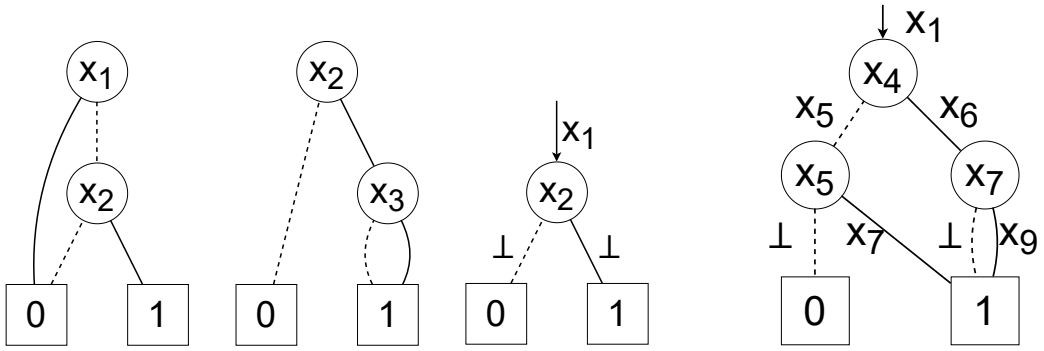
*Tagged binary decision diagrams* (TBDDs) try to merge the reduction rules from BDDs and ZBDDs into one framework. This approach, while being more complex, is more efficient [34]. So, in total, there are four reductions to be applied – merge all terminal nodes $(0, 1)$, merge duplicate nodes, remove nodes whose low and high edge point to the same node, remove nodes whose high edge points to terminal 0 node.

Obtaining a canonical (reduced) form of a TBDD works by first merging terminals, then merging duplicate nodes, then first applying BDD-specific elimination and then applying ZBDD-specific elimination. The name "tagged" comes from the tags that are used on edges between two adjacent nodes of a TBDD. Considering an edge connecting two adjacent nodes $i \to k$, variables $x_i, x_k$ labeling them and the variable order being such that $x_i < x_k$:

- If the tag is a variable $x_j$, such that $x_i < x_j \leq x_k$, then:
  - variables before the tag ($\forall x_r : x_i < x_r < x_j$) were skipped based on the BDD-specific rule ("don't-care" variables),
  - variables after the tag ($\forall x_r : x_j \leq x_r < x_j$) were skipped based on the ZBDD-specific rule ("high-zero" variables).
- If the tag is labeled $\perp$, then all remaining variables were skipped according to the BDD-specific rule ("don't-care" variables).

Every edge leading to a node (even root node and terminal nodes) is labeled with a tag. Edges to internal nodes can only be labeled with a variable, such that its index in the order is greater than source node's and smaller or equal to the target node's. Edges to leaves can be labeled with a variable, or with the $\perp$ tag. Intuitively, to only apply a BDD reduction rule between two nodes labeled $x_i, x_k : x_i < x_k$, the tag is set to $x_k$. To only apply ZBDD reduction rule between two nodes $x_i, x_k : x_i < x_k$, the tag is set to $x_{i+1}$. See Figure 2.7 for a demonstration of TBDD semantics.

TBDDs offer a more compact representation than BDDs in many cases, as measured in [34], and offer similarly high parallel scalability as BDD-based models from the past. However, there are question marks surrounding the flexibility of operations on variable order. Dynamic variable reordering algorithms, such as sifting, can be quite complex in the case of TBDDs' more complex reduction rules.




(a) BDD of $f_5$ (b) ZBDD of $f_5$ (c) TBDD of $f_5$ (d) TBDD of $f_6$

Figure 2.7: Demonstration of TBDDs. Figures 2.7a, 2.7b, and 2.7c depict the representations of a function $f_5(x_1, x_2, x_3) = \neg x_1 \wedge x_2$ using a BDD, ZBDD, and TBDD respectively. Figure 2.7d depicts a TBDD of a function $f_6(x_1, \ldots, x_9) = (\neg x_2 \wedge \neg x_3 \wedge \neg x_4 \wedge x_5 \wedge x_7 \wedge \neg x_8 \wedge \neg x_9) \vee (\neg x_2 \wedge \neg x_3 \wedge x_4 \wedge \neg x_6 \wedge \neg x_7) \vee (\neg x_2 \wedge \neg x_3 \wedge x_4 \wedge \neg x_6 \wedge x_7 \wedge \neg x_9)$, alternatively represented as $S = (*0001*000) \cup (*001*00**) \cup (*001*01*0)$, where conjoined items are sets of bit-vectors with variables on positions with 0s and 1s are fixed as such, and variables on positions with the star symbol can be assigned either 0 or 1, e.g. $(0*) = \{(01), (00)\}$. The Boolean function can be obtained from the TBDD by looking at each path from the root to leaf **1** and assigning False to all variables that are skipped and equal to or greater than the tag. Figures were adapted from [34].

## 2.8 Binary decision diagrams with chain reductions

Chain reductions allow BDDs and ZBDDs to take advantage of each other's reduction rules. So while similar to TBDDs in the essential idea, they differ in the fact that instead of the edge-specified information about reductions, both these models store information about their chain reductions in the nodes. [8]

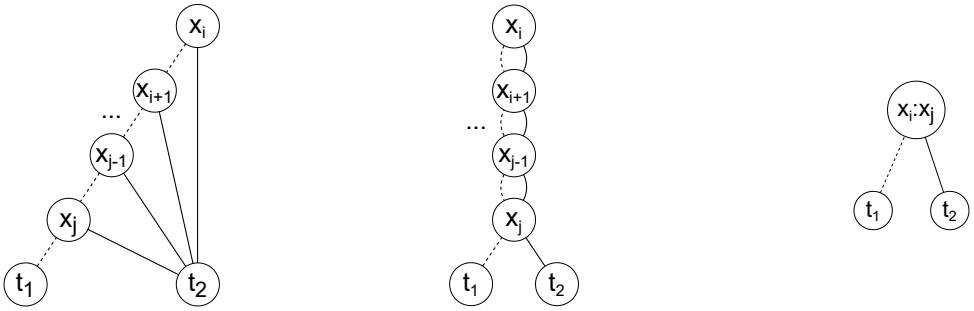
There are two repeating patterns that chain reductions aim to eliminate: OR-chains and DON'T-CARE-chains. In BDDs, a commonly occurring OR-chain consists of a series of connected nodes, each of which has a low edge pointing to the next node in the series and the high edge pointing to some common target of all the nodes in the chain. So OR-chains are essentially a generalized chained version of the pattern ZBDDs initially aimed to eliminate. In ZBDDs, a DON'T-CARE-chain consists of a series of nodes, whose both low and high edge point to the next node in the chain.

- *Chain-reduced ordered binary decision diagrams* (CBDDs) are obtained from a reduced BDD by chaining nodes from each OR-chain into one node.
- *Chain-reduced zero-suppressed binary decision diagrams* (CZDDs) are obtained from a reduced ZBDD by chaining nodes from each DON'T-CARE-chain into one node.

The chain-reduced BDD (CBDD) of a function will be no larger than its BDD representation, and at most three times the size of its CZDD representation. Extensions to the standard algorithms for operating on BDDs and ZDDs enable them to operate on the chain-reduced versions.

The semantics of *chain* nodes thus differ depending on which model is used. Chain nodes are different from general nodes, and are specified by two variables. One variable marks the start of the chain and the other marks its end.

In cases where CBDDs are less efficient than ZBDDs (and vice versa for CZDDs versus BDDs), CZDD representation will be at most twice the size of its BDD representation, while CBDD representation will be at most three times the size of its CZDD representation. CBDD and CZDD cannot be larger than their non-chain-reduced variants. Chain reduction can provide significant benefits in terms of both memory and execution time, as evaluated on benchmarks representing word lists, combinatorial problems and digital circuits [8]. See Figure 2.8 for demonstration of chain reductions.




OR-chain DON'T-CARE-chain Chain-reduction result

Figure 2.8: Generalized versions of OR-chains, found in BDDs, and DON'T-CARE-chains found in ZBDDs. The image on the right is the reduced OR-chain (in CBDDs), while also representing reduced DON'T-CARE-chain (in CZDDs) – the result would be the same for same-length chains. Note that the chain node contains two variables. Figure adapted from [8].

## 2.9 Binary decision diagrams with edge-specified reductions

Following the pattern of integrating multiple reduction rules into one framework, *BDDs with edge-specified reductions* (ESRBDDs) [4] not only merge reduction rules of BDDs and ZBDDs, they also introduce another rule, which mirrors the ZBDD-specific reduction rule, as will be explained shortly. From the name itself it is clear that, similarly to TBDDs, ESRBDDs use edges to specify reduction rules. Nodes themselves do not contain information about reduction, as was the case with CBDDs/CZDDs. ESRBDDs are much easier to understand than TBDDs. While tags in TBDDs hide information about two reductions at once, each edge of an ESRBDD only specifies one reduction:

- `S` (short) – no reduction rule applied on the edge (the variables of source and target should be right next to each other in the variable order).
- `X` ("don't care") – removed redundant nodes. Nodes missing between the source and target of this edge have been removed using the BDD reduction rule.
- $\texttt{H}_0$ ("zero-suppressed" rule) – removed zero-suppressed nodes. Nodes missing between the source and target of this edge have been removed using the ZBDD reduction rule, i.e. their high edge pointed to terminal 0.

- $\mathtt{L_0}$ ("one-suppressed" rule) – removed one-suppressed nodes. Nodes missing between the source and target of this edge have been removed, as they were one-suppressed. One-suppressed nodes are nodes whose low edge points to terminal zero.

An edge $e$ in an ESRBDD is a pair $e = (e.rule,\ e.node)$, where $e.rule \in \{\mathtt{S}, \mathtt{X}, \mathtt{L_0}, \mathtt{H_0}\}$ is a reduction rule used on edge $e$ and $e.node$ is the target node of the edge. An ESRBDD is reduced if it contains no duplicates, no redundant, zero-suppressed, or one-suppressed nodes and edge leading to terminal zero can only have rules from $\{\mathtt{S}, \mathtt{X}\}$. An important aspect of ESRBDDs that differentiates them from models described earlier, is that the reduction rules $\mathtt{X}, \mathtt{H_0}, \mathtt{L_0}$ are not prioritized over each other (TBDDs and CBDDs first apply BDD reductions, then ZBDD/chain reductions, CZDDs first apply ZBDD reductions, then remove DON'T-CARE-chains using BDD reductions). In ESRBDDs, the reduction algorithm works using a deterministic depth-first traversal of the graph structure and applies any of the $\mathtt{X}, \mathtt{H_0}, \mathtt{L_0}$ rule, when it can, until there is no redundant, zero-suppressed, or one-suppressed node left.

Results [4] show that ESRBDDs are more efficient than CBDDs, CZBDDs, or TBDDs. Along with $\mathtt{H_0}$ and $\mathtt{L_0}$ rules, $\mathtt{H_1}$ (high-one) and $\mathtt{L_1}$ (low-one) rules are stated as possible extension to the ESRBDD model.

### Adding complemented edges to ESRBDDs

One of the latest publications in the area of BDD research involves adding complemented edges to ESRBDDs. *BDDs with complemented edges and edge-specified reductions*, or CESRBDDs [3], offer the same advantages as ESRBDDs, but thanks to complemented edges, for each Boolean function $f$ and its negation $\neg f$, represented by two distinct nodes in the CESRBDD, one node does not have to be stored. So for a pair of nodes $u, v$ representing complementing Boolean functions, if $u$ is stored, then all edges leading to $v$ can be redirected to $u$, with a complement bit set to one.

On top of adding complemented edges, CESRBDD also incorporate rules $\mathtt{L_1}, \mathtt{H_1}$ suggested in [4]. The use of complement bits in BDDs was originally introduced by Akers back in 1978 [2], so the idea itself is not new. Each edge in a CESRBDD is represented by a triple $e = (e.rule,\ e.comp,\ e.node)$, where the differences from ESRBDDs are that $e.rule \in \{\mathtt{S}, \mathtt{X}, \mathtt{L_0}, \mathtt{L_1}, \mathtt{H_0}, \mathtt{H_1}\}$, and $e.comp \in \{0, 1\}$ is the complement bit of the edge.

By definition, a CESRBDD is reduced, if:

- It contains no duplicates.
- No edge points to the terminal node **1**. Thanks to complement bits, the terminal node **1** is redundant, edges leading to **0** with complement bit set to 1 can be used instead.
- For every non-terminal node $p$, the complement bit of its low edge is 0.
- For any edge $(\rho, 0, \mathbf{0})$, it holds that $\rho \in \{\mathtt{X}, \mathtt{L_1}, \mathtt{H_1}\}$. For any edge $(\rho, 1, \mathbf{0})$, it holds that $\rho \in \{\mathtt{X}, \mathtt{L_0}, \mathtt{H_0}\}$. Also, rules $\mathtt{L_1}, \mathtt{H_1}$ are not used on edges originating from nodes representing the last two variables. There are special reduction patterns that preserve canonicity for nodes with variables of the last two levels, since there are multiple ways to represent them using combinations of complement bits and different boxes. They are described in detail in [3], but to put it simply, $\mathtt{X}$ is preferred over $\{\mathtt{H_1}, \mathtt{H_0}\}$ and those are preferred over $\{\mathtt{L_0}, \mathtt{L_1}\}$.
- There are no pattern nodes (redundant, high-zero, high-one, low-zero, low-one).

- There is no "$x_1$ and $x_2$" node, i.e. a node at level 2, with the low edge = $(\mathtt{X}, 0, \mathbf{0})$, and the high edge = $(\mathtt{H_1}, 0, \mathbf{0})$.

The conditions for a reduced and canonic CESRBDDs are more complex than their predecessors, but they make CESRBDDs very efficient. Since CESRBDDs in most ways work exactly like ESRBDDs, there is no precedence between the reduction rules. This makes it easier for users, since they do not have to set the order of importance based on the specific application. Nodes of CESRBDDs are more compact than those in CBDDs, CZDDs, and TBDDs (less information stored per node). CESRBDDs themselves also are more compact than other models (less nodes required to represent functions), as shown by experimental results in [3].

# Chapter 3

# Manipulating Binary Decision Diagrams — Apply

The previous chapter did not go in depth into the important operations over BDDs, only mentioned them. This chapter is going to provide an overview for definitions and implementations of the most important operations on Boolean functions (and their BDD representations). Since the goal of the thesis is implementing the Apply operation, the main focus of the description will be on that particular function – complexities, explanations, different techniques of optimization.

## 3.1 Important operations on binary decision diagrams

A compact and canonic representation of a Boolean function is important, but not enough in most applications. Usually it is important to be able to perform operations with Boolean functions and manipulate them without loosing canonicity, while still maintaining good performance. What follows are brief definitions with intuitive descriptions for all important operations on Boolean functions. The BDD implementations provided are based on the descriptions in [9].

### Evaluation

Given a Boolean function $f(x_1, \ldots, x_n)$ and a vector of truth values $\mathbf{a} = (a_i, \ldots, a_n)$ where each $a_i \in \{0, 1\}$, the task is to return $f$ applied to $\mathbf{a}$, or $f(\mathbf{a}) \in \{0, 1\}$. Intuitively, this is just a lookup in the truth table of a function.

Implementing evaluation on basic BDDs is straightforward. Given a vector of truth values (the vector order conforms to the variable order) and a BDD, the algorithm just traverses one path of the BDD. The next visited node $v_n$ is picked by the value of the vector element corresponding to the index of the variable of the current node $v_c$. If the entry is zero, $v_n = low(v_c)$, if one, $v_n = high(v_c)$. This is repeated until a terminal node $v_t$ is reached, then $val(v_t)$ is returned. The upper bound time complexity for evaluation is $O(n)$ where $n$ is the number of variables.

### Restriction

Given the Boolean function $f$ over a vector of Boolean variables $(x_1, x_2, \ldots, x_n)$, the restriction of $f$ where some $x_i$ is replaced by a truth value $b$, is the following Boolean function:

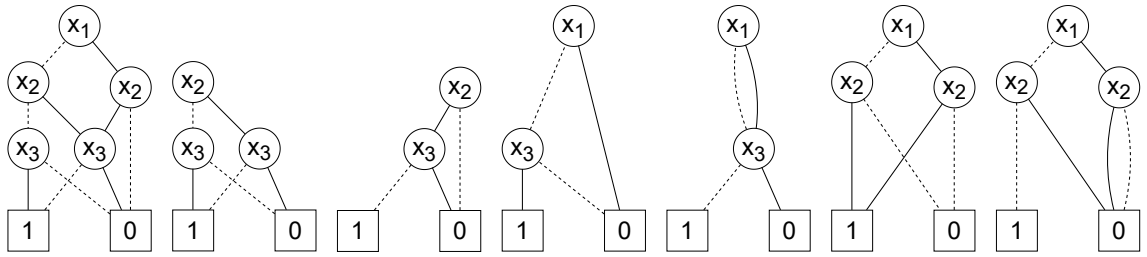
$$f|_{x_i=y}(x_1, x_2, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n) \ .$$

Note that there are two restrictions of function $f$ with respect to variable $x_i$ (for the two truth values). These functions are called *cofactors* of $f$ with respect to $x_i$.

The BDD implementation of the restriction $f|_{x_i=b}$ works by traversing the graph structure, finding edges pointing to such nodes $v$ where $var(v) = x_i$ and redirecting them to the node $v'$ such that $v' = low(v) \iff b = 0 \ \wedge \ v' = high(v) \iff b = 1$ holds. Since references to nodes with the restricted variable can be multiple and anywhere in the graph structure, this algorithm works in $O(|G|)$ where $|G|$ is the number of nodes in BDD $G$. The result of restriction can leave some redundant or duplicate nodes in the BDD structure (see Table 3.1 and Figure 3.1 for demonstration), so calling reduction algorithm after restriction is necessary to preserve canonicity.

Table 3.1: Truth table for function $f_1$ and its restrict operations. The boxed values represent relevant results for the cofactors.

| $x_1$ | $x_2$ | $x_3$ | $f_1$ | $f_1\|_{x_1=0}$ | $f_1\|_{x_1=1}$ | $f_1\|_{x_2=0}$ | $f_1\|_{x_2=1}$ | $f_1\|_{x_3=0}$ | $f_1\|_{x_3=1}$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | [0] | 0 | [0] | 1 | [0] | 1 |
| 0 | 0 | 1 | 1 | [1] | 0 | [1] | 0 | 0 | [1] |
| 0 | 1 | 0 | 1 | [1] | 1 | 0 | [1] | [1] | 0 |
| 0 | 1 | 1 | 0 | [0] | 0 | 1 | [0] | 1 | [0] |
| 1 | 0 | 0 | 0 | 0 | [0] | [0] | 1 | [0] | 0 |
| 1 | 0 | 1 | 0 | 1 | [0] | [0] | 0 | 0 | [0] |
| 1 | 1 | 0 | 1 | 1 | [1] | 0 | [1] | [1] | 0 |
| 1 | 1 | 1 | 0 | 0 | [0] | 0 | [0] | 1 | [0] |




(a) $f_1$ (b) $f_1|_{x_1=0}$ (c) $f_1|_{x_1=1}$ (d) $f_1|_{x_2=0}$ (e) $f_1|_{x_2=1}$ (f) $f_1|_{x_3=0}$ (g) $f_1|_{x_3=1}$

Figure 3.1: Demonstration of the restriction on a Boolean function. Function $f_1(x_1, x_2, x_3)$ is the same as in Figure 2.1, and Table 3.1. (a) is its BDD representation followed by intermediate results of restriction (b-g) before applying reduction algorithm. Notice how (e) and (g) contain redundant nodes, and (f) has duplicate nodes (labeled $x_2$).

**Shannon expansion** [31], an important concept of Boolean functions, can be defined using restriction. Shannon expansion of a function w.r.t. variable $x_i$ is given by:

$$f = (x_i \wedge f|_{x_i=1}) \ \vee \ (\neg x_i \wedge f|_{x_i=0})$$

Shannon expansion creates a new function by arbitrarily adding another variable to another existing function. This will be important later for applying Boolean operators on two Boolean functions, and for specifying operations like composition.

### Composition

Composition is when an argument $x_i$ of a function $f(x_1, \ldots, x_n)$ is replaced by the result of another function $g(x_1, \ldots, x_n)$, such a composition is then denoted $f|_{x_i=g}$. Mathematically:

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

Note that a function can even be composed into itself, $f|_{x_i=f}$. There is an alternative way of express the composition of some function, using Boolean operations and restriction. This expression intuitively gives an indication as to how to perform composition on BDDs, if procedures for Apply and restriction are given:

$$f|_{x_i=g} = (g \wedge f|_{x_i=1}) \vee (\neg g \wedge f|_{x_i=0})$$

Table 3.2: Table of all compositions between Boolean function examples $f_1$ and $f_2$

| $x_1$ | $x_2$ | $x_3$ | $f_1$ | $f_2$ | $f_1\|_{x_1=f_2}$ | $f_1\|_{x_2=f_2}$ | $f_1\|_{x_3=f_2}$ | $f_2\|_{x_1=f_1}$ | $f_2\|_{x_2=f_1}$ | $f_2\|_{x_3=f_1}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |

### Satisfiability

Given a function $f$, satisfiability checking procedure returns true if and only if there is an assignment of truth values to variables such that the function returns true. In BDDs, this question is basically the following: Is the terminal node **1** reachable? Additionally to the yes/no answer, satisfy often requires a vector of truth values describing the specific variable assignment for which the function is true.

In reduced BDDs, there is only one BDD for which the function is false and it only consists of the terminal node **0**. For every other node, the **1** terminal is reachable. Thus, checking satisfiability (without requiring the assignment) in reduced BDDs can be done in $O(1)$ by checking if the root node of the reduced BDD is not **0**. If the assignment is also required, then the algorithm works in $O(n)$ and employs simple backtracking. It is only necessary to backtrack one step back to node $v$ and that is when the terminal **0** is reached. Then the other branch of $v$ is picked for which it is guaranteed that **1** is reachable. Then procedure keeps a vector of truth values during the traversal of the tree, editing it as needed and filling it with zeros (for example) in case of edges that skip variables.

### Apply

Given two Boolean functions $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_n)$ and a (binary) Boolean operator $\odot$, return a function $h(x_1, \ldots, x_n)$, such that $\forall(x_1, \ldots, x_n) \in \{0,1\}^n$:

$$h(x_1, \ldots, x_n) = (f \odot g)(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \odot g(x_1, \ldots, x_n)$$

In the following, Apply is considered only with binary operators, since all other functions can be represented using connectives from some functionally complete set. The only missing ingredient is negation, which is unary, but in reduced BDDs, negation can be done in constant time by simply switching terminal **0** and terminal **1** nodes. But even if it were not, negation of $f$ can be obtained as a result of $f \oplus \mathbf{1}$. Another important use case of Apply is testing for implication, i.e. checking if $f_1 \wedge \neg f_2 = \mathbf{0}$.

## 3.2 Basic overview of Apply algorithm in BDDs

Given two Boolean functions $f, g$ and a binary Boolean operator $\odot$, the algorithm starts at the root nodes of the two BDDs representing $f, g$ and recursively descends down in the graphs, while building a resulting BDD $G$ representing the result. The nodes of the result are created at the branching points of the two input graphs. The control flow of the algorithm is based on the recursion derived from the Shannon expansion, i.e.:

$$f \odot g = (\neg x_i \wedge (f|_{x_i=0} \odot g|_{x_i=0})) \vee (x_i \wedge (f|_{x_i=1} \odot g|_{x_i=1}))$$

Intuitively, the left disjunct represents recursing down on the low edges of inputs, the right disjunct represents recursing down on the high edges of inputs. For the recursion to work properly, the nodes of the both input BDDs have to represent the same variable.

Since reduced BDDs can have edges that skip variables, the equality of node variables is not guaranteed during the simultaneous recursive descent. The following presents the three different cases of what can happen when nodes $u$ from the first input BDD and $v$ from the second input BDD are visited during the recursive Apply calls:

1. Both $u$ and $v$ are terminal nodes. Then the resulting BDD is the terminal node $val(u) \odot val(v)$. In the following two cases, suppose at least one of $u, v$ is not a terminal node.

2. $var(u) = var(v) = x_i$. Then a node $w$ with variable $x_i$ is created, and two recursive calls are invoked. Apply called on nodes $low(u), low(v)$ will create a BDD rooted in $w_L$, and this BDD will be pointed to by $low(w) = w_L$. Apply called on nodes $high(u), high(v)$ creates $w_H$, and $high(w) = w_H$.

3. $var(u) < var(v) \ \vee \ v \in \{\mathbf{0}, \mathbf{1}\}$ (one node has a greater variable or is a terminal, note that the root node has the "smallest" variable). In that case, recursion only descends with regards to the node $u$, the second argument node stays the same, until either the variables are the same or both nodes represent terminal nodes, so that algorithm can proceed as in the cases 1 or 2. This case is mirrored when the conditions for $u, v$ are reversed, i.e. $var(u) > var(v) \ \vee \ u \in \{\mathbf{0}, \mathbf{1}\}$.

In the third case, a new node is virtually introduced in the resulting BDD, this phenomenon is sometimes called *node materialization*. Notably, the materialized node is not

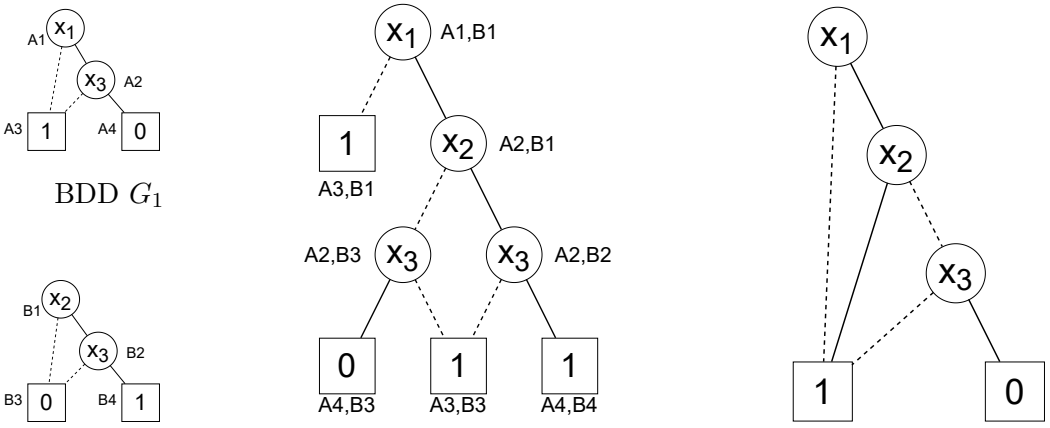
introduced to any of the inputs, however, it can be a part of the result. Materialized nodes are kept in the call stack of the recursion. If only the first two cases were applied, the resulting BDD would be of the same size and shape as the two arguments (which also have to be the same size for this to be possible), since the argument would be isomorphic (up to the values in terminal nodes). Regardless of which cases are applied during the algorithm, the resulting graph is not guaranteed to be reduced/canonical. That is why after the top-level call (on the roots of the initial functions) of Apply is over, the reduction of the result has to take place.

### Optimizing the complexity with memoization

A naive implementation of Apply would have an exponential time complexity (wrt. the number of variables $n$), since every Apply call generates two recursive calls (when one of the nodes is a non-leaf). Using the following three modifications can reduce this complexity.

First, many times the Apply function is called on the same arguments. By utilizing memoization of Apply calls, or *call cache*, the upper bound for complexity drops from exponential wrt. the number of variables to $O(|G_1| \cdot |G_2|)$ where $|G_1|, |G_2|$ are the number of nodes in BDDs $G_1$ and $G_2$. The number of calls cannot be larger than the product of the two sizes, because the theoretical worst case is that for every node $u$ from $G_1$, Apply is called with every node $v$ from $G_2$. Memoization is used extensively in dynamic programming.

Second technique utilizes annihilation/absorption properties of **0** wrt. conjunction and **1** wrt. disjunction. If the function $\odot$ is logical OR, then when one node is terminal **1**, no recursion or materialization is necessary, and **1** can be returned. The same concept applies for AND and **0**.




BDD $G_1$ BDD $G_2$ $Apply(\vee, G_1, G_2)$ before reduction. $Apply(\vee, G_1, G_2)$ after reduction.

Figure 3.2: Demonstration of a run of the BDD Apply algorithm. The figure is adapted from the original BDD paper [9]. Nodes are labeled with arbitrary identifiers to clarify the recursion process. Terminal node labeled $(A3, B1)$ is obtained utilizing the annihilation properties of 1 wrt. function $\vee$. Call cache utilization is demonstrated by $(A2, B3)$ and $(A2, B2)$ pointing to the same *low* node (because it is the result of the same call $(A3, B3)$), since from that point they both called Apply on the same node. If node cache (or unique table) was used, $(A3, B1), (A3, B3), (A4, B4)$ would be the same node. That would also demonstrate the need for reducing after Apply, since $(A2, B2)$ would become redundant (as both its *low* and *high* targets would be the same node before returning from recursion).

The third technique uses node memoization, sometimes called a *unique table*, which is a map of triples $(x_i, id_1, id_2)$ to node $v$, such that $var(v) = x_i$, $id(low(v)) = id_1$, $id(high(v)) = id_2$. Using this cache, anytime a new node has to be created (when returned as a result), it is checked against this cache and if a hit occurs, the node is not created, instead a reference to the cache is used. This way, there will be no duplicate nodes after Apply is finished. Redundant nodes can be checked before inserting the node into the cache, if $id(low(v)) = id(high(v))$ holds, then $low(v)$ is returned instead of $v$. Apply then does not require explicit reduction, because it is done on-the-fly. Using this is correct, because the cache is filled with nodes in a bottom-up order, which consequently checks against all so far created canonical representations of sub-functions. Sometimes in algorithmic notation this technique is omitted, since it is functionally equivalent to using reduce after Apply. See Figure 3.2 for a demonstration of the Apply algorithm. See Algorithm 1 for a pseudocode overview of how one recursive Apply call looks like. The top-level *Apply*() would only create an empty call cache, call *ApplyStep*() on root nodes of BDDs, and after it is done, reduce the result.

**Algorithm 1:** BDD version of $ApplyStep(\odot, u, v)$

**Input:** Binary Boolean operator $\odot$, BDD node $u$ representing function $f_u$, BDD node $v$ representing function $f_v$

**Result:** BDD node $n$ representing function $f_u \odot f_v$

1 **if** $u \in \{\mathbf{0}, \mathbf{1}\} \wedge v \in \{\mathbf{0}, \mathbf{1}\}$ **then return** $u \odot v$ ;

`// Checking if annihilation laws can be applied`

2 **if** $(\odot = \text{OR}) \ \wedge \ (u = \mathbf{1} \vee v = \mathbf{1})$ **then return 1** ;

3 **if** $(\odot = \text{AND}) \ \wedge \ (u = \mathbf{0} \vee v = \mathbf{0})$ **then return 0** ;

`// Checking if absorption laws can be applied`

4 **if** $((\odot = \text{AND} \wedge u = \mathbf{1}) \vee (\odot = \text{OR} \wedge u = \mathbf{0}))$ **then return** $v$ ;

5 **if** $((\odot = \text{AND} \wedge v = \mathbf{1}) \vee (\odot = \text{OR} \wedge v = \mathbf{0}))$ **then return** $u$ ;

6 **if** `callCache`$[(u, v)] = t \ \wedge \ t \neq \perp$ **then return** t ;

7 $x_n \leftarrow \min\{var(u), var(v)\}$;

8 $l_n \leftarrow ApplyStep(\odot, \ \ var(u) = x_n \ ? \ u : low(u), \ \ var(v) = x_n \ ? \ v : low(v))$;

9 $h_n \leftarrow ApplyStep(\odot, \ \ var(u) = x_n \ ? \ u : high(u), \ \ var(v) = x_n \ ? \ v : high(v))$;

`// Here could be a check of redundant node and of duplicate node against the node cache / unique table`

10 create a new node $n$, such that $var(n) = x_n$, $low(n) = l_n$, $high(n) = h_n$;

11 `callCache`$[(u, v)] \leftarrow n$;

12 **return** $n$

## 3.3 Apply in modifications of BDDs

In ZBDDs, Apply works similarly, but in general it must recurse one level at a time. Since the "long" edges (i.e. edges between nodes labeled $x_i, x_j$, where $i+1 \neq j$) in BDDs represent variables that are irrelevant to the result, variable skipping in BDD Apply works fine. But in ZBDDs, long edges work differently, as discussed in Section 2.6. Only some operators can utilize level skipping. Specifically, operator $\odot$ must satisfy $0 \odot 0 = 0$, so that applying to long edges pointing to $u$ and $v$ returns a long edge to $u \odot v$. [3, 25],

The paper introducing TBDDs [34] did not provide a full description of the Apply algorithm, only how cofactors are computed. The cofactor routine systematically checks whether the node's edges were not reduced according to the reduction rules – first BDD reduction, then ZBDD reduction, then a special case of reduction applied when nodes matching both previous reductions alternate that ensures an extra node is created for canonicity such that ZBDD rule is enforced maximally. After establishing that no reduction rule was used, the successors of the node are returned.

The Apply algorithm in CBDDs and CZDDs [8] only modifies the variable level splitting step, the high and low cofactor generation step and how the results after Apply are combined to determine the result node parameters.

In CBDDs, based on the splitting levels of a chain, one of the following three cases can happen wrt. computing the cofactors $low(v_i, (x_i : x_j))$:

- Both cofactors are the same node $v_i$ – when splitting occurs "above" the node and because in CBDDs, level-skipping edges encode DON'T CARE chains.
- Splitting spans the same nodes as the chain node $v_i$ – cofactors are thus the descendants of the node $v_i$.
- Splitting occurs "below" $v_i$ – a new node is constructed, spanning the remaining part of the encoded OR-chain for the low cofactor, high stays the same.

After recursive Apply calls, the generated nodes are combined to form a result node based on the following two reduction rules (if not applicable, use the results of the recursive calls directly):

- If high and low cofactors are identical, a classic BDD reduction can be applied.
- Chain compression can be applied to create a node absorbing the low cofactor.

In CZDDs, the process is very similar. When splitting spans above the chain variables, the low cofactor is the original node and high cofactor is **0**. When splitting happens at the same levels as the chain node, cofactors are given by the outgoing node's edges.

When combining the results of Apply calls in CZDDs, one of the following three reduction rules can be used (otherwise use the results directly):

- Resulting nodes allow for direct application of ZBDD-rule, i.e. the high edge points to **0**, result is redirected to the low edge.
- ZBDD-rule can be applied, but a node has to be constructed encoding a DON'T-CARE chain on skipped levels.
- Chain compression can be applied to create a node that absorbs the low cofactor.

Apply calls in ESRBDDs and CESRBDDs [3] exhaustively check for all possible combinations of reduction rules, complement bits and edge targets (whether or not it is some specific terminal node) that can eliminate recursion (similar to how absorption/annihilation laws with regards to $\wedge, \vee$ are checked in BDDs). The interplay of reduction rules $\{\mathtt{X}, \mathtt{L}_0, \mathtt{L}_1, \mathtt{H}_0, \mathtt{H}_1\}$, complement bits on edges, and terminal node values can lead to many non-canonical CESRBDD (after Apply) that can be simplified, so the initial checks do that. If no "recursion eliminating" pattern is found, CESRBDD Apply tries to look into the call cache, which, similarly to ESRBDDs, contains the two edges, so on top of nodes and reduction rules (as in ESRBDDs), complement bits also differentiate unique nodes. The ternary

operator used in Algorithm 1 (lines 6 and 7) is an example of a cofactor routine (cofactor is a term used extensively in descriptions of algorithms in [3]) that resolves what combination of nodes should the apply call be made on. When recursively calling Apply on CESRBDDs, the "cofactor" routine performs additional checks:

- When a node materialization is needed, based on the complement bit of parent node, it sets the complement bit of the edge returned by the Apply call.
- When recursing through a low edge and encountering $\mathtt{L_t}$ rule, set the result edge (since the call "triggered" the reduction) to a constant function $(\mathtt{X}, \mathtt{t}, \mathbf{0})$. Similarly for high edge and $\mathtt{H_0}, \mathtt{H_1}$
- Makes sure that the recursive call is made on a short edge.

In ESRBDDs, this routine is very similar, just omitting working with complement bits, and not considering $\mathtt{L_1}, \mathtt{H_1}$ rules in the patterns.

# Chapter 4

# Automata-based Binary Decision Diagrams — ABDDs

Automata-based binary decision diagrams (ABDDs) were introduced as an alternative way of compactly representing binary decision diagrams [24]. This chapter contains compacted preliminary definitions and explanations of concepts crucial to understanding this conceptual framework and draws heavily from Chapter 4 of [24] that first introduced ABDDs and their canonization algorithms. Important tree automata concepts and definitions were taken from [12] and are explained in more detail there, while the tree automata notation is inspired by [1]. For more detailed explanation with examples, refer to the original thesis [24].

Firstly, tree automata and their properties are introduced. Then, some key concepts of ABDDs are introduced, such as alphabets used during different parts of canonization, boxes, trees. Finally, canonization algorithms – unfolding, normalization, folding – are explained.

## 4.1 Tree automata preliminaries

The usual notion of formal languages (from Chomsky's hierarchy) refers to sets of strings, each string consisting of symbols from some finite non-empty set called an *alphabet*. Tree languages are sets of *trees*, and each node/vertex of the tree is labeled with some symbol from a *ranked alphabet* $\Sigma$.

A *ranked alphabet* is a finite set of symbols $\Sigma$ with a ranking function (arity-assigning function) $\# : \Sigma \to \mathbb{N}$. Rank, or *arity*, of a symbol $\#(a) \in \mathbb{N}$ is a fixed number of child nodes for any tree node labeled with $a$. Symbols with rank 0 are leaf symbols and each ranked alphabet should contain at least one such symbol, otherwise the trees could not be "terminated" and would be infinite. $\#(a) = k$ denotes that symbol $a$ has arity $k$ (alternatively, it can also be denoted as $a_{/k}$).

A *tree* over a ranked alphabet is defined as a partial mapping $t : \mathbb{N}^* \rightharpoonup \Sigma$, where $\mathbb{N}^*$ are sequences of numbers which represent positions in some theoretical infinite tree (every branch has length $\omega$, and every node has $\omega$-many children). $\varepsilon$ is the root node, 00 is the left-most descendant of the left-most descendant of the root, etc. Since this mapping is partial, only some sequences/positions will be present in the tree (thus making it finite).

The *domain of a tree* $\mathrm{dom}(t)$ is a set of sequences from $\mathbb{N}^*$, each sequence representing some tree *node*. Formally, for a tree $t : \mathbb{N}^* \rightharpoonup \Sigma$, $\mathrm{dom}(t)$ has to satisfy the following:

1. $\mathrm{dom}(t)$ is a finite, non-empty, prefix-closed subset of $\mathbb{N}^*$.

2. $\forall v \in \mathrm{dom}(t) : \#(t(v)) = n > 0 \Rightarrow \{i \mid v.i \in \mathrm{dom}(t)\} = \{0, \ldots, n-1\}$

3. $\forall v \in \mathrm{dom}(t) : \#(t(v)) = 0 \Rightarrow \{i \mid v.i \in \mathrm{dom}(t)\} = \emptyset$

Note that . denotes concatenation of strings in this case. Non-emptiness of $\mathrm{dom}(t)$ always holds because $\varepsilon \in \mathrm{dom}(t)$ (root node). Prefix-closure of $\mathbb{N}^*$ naturally holds because every node along some branch of the tree belongs to the same tree. The second condition ensures correct labeling of child nodes of each inner node of a tree. The third condition ensures that leaf nodes do not have any child nodes.

The $i$-th child of a node $v$ is node $v.i$ and the $i$-th *subtree* of $v$ is defined as a tree $t'$ such that $\forall v' \in \mathbb{N}^* : t'(v') = t(v.i.v')$. The set of all trees $\mathcal{T}_\Sigma$ over the alphabet $\Sigma$ is called the universal tree language over $\Sigma$. The height $|t|$ of a tree $t \in \mathcal{T}_\Sigma$ rooted in position $p$ is inductively defined as $|t| = 1$ iff $p$ is a leaf position, otherwise $|t| = 1 + \max\{|t(p.i)| \mid i \in \{0, \ldots, n-1\}\}$.

### Tree automaton definition

A nondeterministic finite tree automaton (NFTA) is a tuple $\mathcal{A} = (Q, \Sigma, \Delta, F)$ where:

- $Q$ is a finite set of states,
- $\Sigma$ is a ranked alphabet,
- $F \subseteq Q$ is a set of final states,
- $\Delta$ is a finite set of transitions of the form $((q_1, \ldots, q_n), a, q)$ where $q, q_1, \ldots, q_n \in Q \wedge a \in \Sigma \wedge \#(a) = n$. Transitions of the form $((), a, q)$ are called *leaf transitions*.

A *run of a tree automaton* $\mathcal{A} = (Q, \Sigma, \Delta, F)$ over a tree $t \in \mathcal{T}_\Sigma$ is a mapping $\rho : \mathrm{dom}(t) \to Q$, such that for each position $v \in \mathrm{dom}(t)$ where $t(v) = a$ for some $a \in Sigma, \#(a) = n$, and $q = \rho(v)$, it holds that if $q_i = \rho(v.i)$ then $((q_1, \ldots, q_n), a, q) \in \Delta$ for some $1 \leq i \leq n$.

The existence of a run $\rho$ of $\mathcal{A}$ over $t$ is denoted by $t \overset{\rho}{\Longrightarrow} q$, where $\rho(\varepsilon) = q$. A run is *accepting* if $\rho(\varepsilon) = q \in F$. Intuitively, a run is a tree labeled with states from $Q$ that respects the arities of symbols in the initial tree and transitions from $\Delta$. A language accepted by a state $q$ of a tree automaton is defined as $\mathcal{L}(\mathcal{A})(q) = \{t \mid t \overset{\rho}{\Longrightarrow} q)$. A *language recognized by a tree automaton* $\mathcal{A}$ is $\mathcal{L}(\mathcal{A}) = \bigcup_{q \in F} \mathcal{L}(\mathcal{A})(q)$.

### Tree automata properties

The previous definition refers to nondeterministic bottom-up tree automaton. In a NFTA, there can possibly exist more than one run over the same tree. *Deterministic* finite tree automata (DFTA) are a special case of NFTA, in which such ambiguity cannot occur, as there are no two transitions with the same left-hand side. A DFTA constructed from an equivalent NFTA of $n$ states, in the worst case, can have $2^n$ states. DFTAs and NFTAs are equivalent in their expressive power – for every language $\mathcal{L}$ recognized by a NFTA, there exists a DFTA recognizing $\mathcal{L}$. The class of languages recognized by DFTAs/NFTAs is called *regular tree languages*.

A NFTA $\mathcal{A}$ is *complete* if there is at least one rule $((q_1, \ldots, q_n), a, q) \in \Delta$ for all $a \in \Sigma, \#(a) = n$ and $q_1, \ldots, q_n \in Q$. A state $q$ is *accessible*, or reachable iff $\exists t : t \overset{\rho}{\Longrightarrow} q$. NFTA is *reduced* if all its states are accessible. For practical reasons, it is more convenient to work with reduced NFTAs. A state $q$ is *useful* in $\mathcal{A}$ if $\rho(p) = q$, where $\rho$ is an accepting run of $\mathcal{A}$

on some tree $t \in \mathcal{L}(\mathcal{A})$ and $p \in \mathrm{dom}(t)$. In other words, state $q$ is useful in an automaton, if during an accepting run of a tree from the automaton's language, the state $q$ is visited. NFTA is *trimmed*, if it contains no useless states.

The class of regular tree languages (recognized by bottom-up NFTAs/DFTAs), is closed under *union*, *complementation*, and *intersection*.

Bottom-up tree automata compute a run of the tree starting from leafs (states that have some leaf transition), trying to build their way to a some root state $q \in F$. Another, more intuitive, way of performing a run over a tree starts from the root node of a tree and tries to build the tree top-down, recursively. TAs working in this way are called *top-down* tree automata.

A nondeterministic *top-down* finite tree automaton is a tuple $\mathcal{A} = (Q, \Sigma, \Delta, \mathcal{R})$, differing from the bottom-up NFTA definition in these two ways:

- $R \subseteq Q$ is a set of initial states called *root states*, and
- rules in $\Delta$ have the form $(q, a, (q_1, \ldots, q_n))$ where $q, q_1, \ldots, q_n \in Q, a \in \Sigma, \#(a) = n$.

The run of a top-down tree automaton is accepting if $\rho(\varepsilon) \in \mathcal{R}$. The expressive power of bottom-up and top-down nondeterministic tree automata is equivalent. However, deterministic top-down tree automata are weaker than deterministic bottom-up tree automata. In other words, for some top-down tree automata, there is no deterministic equivalent. The class of languages recognized by deterministic top-down tree automata is called *path-closed* tree languages.

## 4.2 ABDD preliminaries

During different phases of ABDD canonization, different alphabets are used. Let $\mathbb{X} = \{x_1, \ldots, x_n\}$ be a finite set of *variables* ordered $x_1 < \ldots < x_n$. The following alphabets contain $k$-ary symbols $\langle \mathtt{L}\colon a, \mathtt{H}\colon b, \ldots \rangle_{/k}$ denoting the function $\{\mathtt{L} \mapsto a, \mathtt{H} \mapsto b, \ldots\}$ where $a, b$ denote some edge type. The ellipsis $\ldots$ denotes additional elements, such as an assigned variable $\langle \mathtt{L}\colon \mathtt{S}, \mathtt{H}\colon \mathtt{S}, \mathit{var}\colon x_i \rangle_{/2}$ ($\mathtt{S}$ denotes short edge, i.e. no reduction rule used). $-\boxed{\langle \mathtt{L}\colon \mathtt{S}, \mathtt{H}\colon \mathtt{S} \rangle}\!\!\rightarrow$ will be used as an alternative notation for the symbols of the following alphabets.

$\Sigma_{\mathtt{LH}} = \{\langle \mathtt{L}\colon \mathtt{S}, \mathtt{H}\colon \mathtt{S} \rangle_{/2}, \mathbf{0}_{/0}, \mathbf{1}_{/0}\}$ is the *low-high alphabet* which serves as the base for all other alphabets. Note that no variables are used here.

$\Sigma_{\oplus} = \Sigma_{\mathtt{LH}} \cup \{\oplus_1, \ldots, \oplus_m \mid m \in \mathbb{N}\}$ is the *port alphabet*. All new symbols have arity 0 and denote *output ports* of boxes (tree automata representing reduction rules).

$\Sigma_{\mathbb{X}} = \{\langle \mathtt{L}\colon \mathtt{S}, \mathtt{H}\colon \mathtt{S}, \mathit{var}\colon x_i \rangle_{/2}, \mathbf{0}_{/0}, \mathbf{1}_{/0}\}$ (for some $x_i \in \mathbb{X}$) is the *binary decision tree alphabet*, enhancing $\Sigma_{\mathtt{LH}}$ with information about variables. Note that symbols from $\Sigma_{\mathbb{X}}$ cannot be used to label looping transitions (variables cannot be repeated on a path of a BDD). Additionally, *.var* notation will refer to the variable $x_i$ used in the symbols with non-zero arity.

$\Sigma_{\mathtt{LH}\mathbb{X}} = \Sigma_{\mathtt{LH}} \cup \Sigma_{\mathbb{X}}$ is used in UBDAs (unreduced binary decision automata), where some transitions come from the BDD-like structure, where transitions are labeled with variables, and some transitions come from the tree automata representing reduction rules (no variables).

The *box alphabet* $\Sigma_\Gamma = \{\langle \mathtt{L}\colon a, \mathtt{H}\colon b, \mathit{var}\colon x_i \rangle_{/k}, \langle \mathit{in}\colon c \rangle_{/l}, \mathbf{0}_{/0}, \mathbf{1}_{/0}\}$, where $a, b, c \in \Gamma, x_i \in \mathbb{X}, k = \#(a) + \#(b), l = \#(c)$ and $\Gamma$ is a set of boxes. $\#(a), \#(b), \#(c)$ refer to port arity of boxes (this will be explained later). $\Sigma_\Gamma$ is used in TAs that can directly be translated

to ABDDs. During the process of canonization, the symbols from the underlying UBDA automaton-like structure get mixed with symbols from this alphabet, which leads to the final alphabet $\Sigma_{\mathtt{LH}\Gamma} = \Sigma_\Gamma \cup \Sigma_{\mathtt{LH}}$.

### Boxes

Let $\mathcal{B} = (Q_\mathcal{B}, \Sigma_\oplus, \Delta_\mathcal{B}, \mathcal{R}_\mathcal{B})$ be a top-down tree automaton, let $\#(\mathcal{B})$ denote the *arity* of the box $\mathcal{B}$ such that $\#(\mathcal{B}) = |\mathcal{M}_\mathcal{B}|$ where $\mathcal{M}_\mathcal{B} = \{\oplus_i : \exists q – \boxed{\oplus_i} \rightarrow () \in \Delta_\mathcal{B}\}$ (i.e. the arity of $\mathcal{B}$ is the number of different ports used in the transition relation of $\mathcal{B}$), and for a tree $t \in \Sigma_\oplus$, let $\oplus_i \in t$ denote that $\exists p \in \mathrm{dom}(t) : t(p) = \oplus_i$.

1. $\mathcal{L}(\mathcal{B}) \neq \emptyset$ (the language of a box is not empty),
2. $\mathcal{B}$ is trimmed,
3. $\forall t \in \mathcal{L}(\mathcal{B}) : \oplus_i \in t \Rightarrow (\forall 1 \leq j < i : \oplus_j \in t) \wedge (\forall t' \in \mathcal{L}(\mathcal{B}) : \oplus_i \in t')$ (this condition ensures port consistency, i.e. every tree from $\mathcal{L}(\mathcal{B})$ uses the same set of ports),
4. $\mathcal{R}_\mathcal{B} = \{r_\mathcal{B}\}$ (a box has one root state),
5. $\forall 1 \leq i \leq \#(\mathcal{B}) : (r_\mathcal{B}, \oplus_i, ()) \notin \Delta_\mathcal{B}$ (i.e. there are no trivial trees in $\mathcal{L}(\mathcal{B})$ such that the root position of the tree is labeled with $\oplus_i$),
6. $\forall 1 \leq i \leq \#(\mathcal{B}) : (\exists! q \in Q_\mathcal{B} : (q, \oplus_i, ()) \in \Delta_\mathcal{B})$ (for every port $\oplus_i$, there is a unique state with an outgoing $\oplus_i$-transition), and
7. Let $var : \{root, \oplus_1, \ldots, \oplus_{\#(\mathcal{B})}\} \rightarrow (\mathbb{X} \cup \{x_{n+1}\})$ be a mapping of root node and all output ports of $\mathcal{B}$ to variables ($x_{n+1} \notin \mathbb{X}$ is a pseudo-variable for *leaf level*), and let $t$ be a tree over $\Sigma_\oplus$. $var$ can be mapped to $t$ if there exists a function $var_t : \mathrm{dom}(t) \rightarrow (\mathbb{X} \cup \{x_{n+1}\})$, such that:
   - $var_t(\varepsilon) = var(root)$,
   - $var_t(p.a) = x_{i+1} \iff var_t(p) = x_i$ for $a \in \{\mathtt{L}, \mathtt{H}\}$, and
   - $var_t(p) = var(\oplus_i)$ for $t(p) = \oplus_i$.

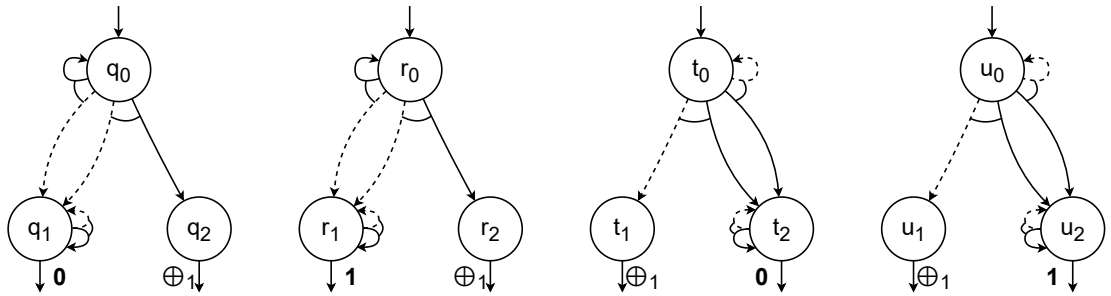
   It is then required that for every mapping $var : \{root, \oplus_1, \ldots, \oplus_{\#(\mathcal{B})}\} \rightarrow (\mathbb{X} \cup \{x_{n+1}\})$, it holds that there is at most one $t \in \mathcal{L}(\mathcal{B})$, such that $var$ can be mapped to $t$.

$\exists!$ denotes the unique existential quantifier ("exists exactly one"). These conditions ensure that boxes are constructed in a way that allows for exactly one interpretation of their semantics in an ABDD (with regards to how Boolean functions are evaluated). Currently, ABDDs utilize 7 boxes, see Figure 4.1.

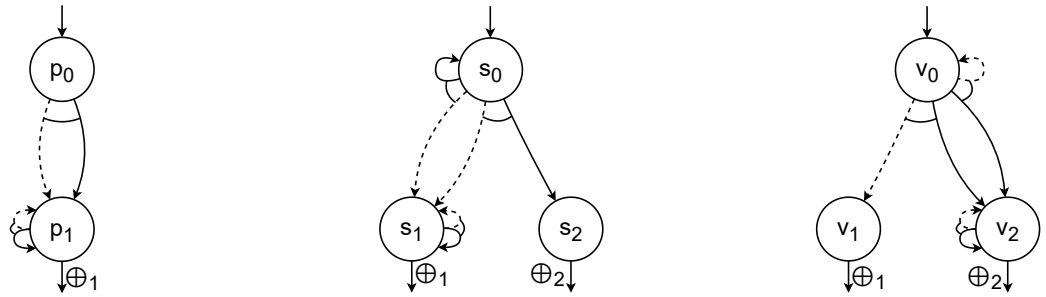
### Binary decision trees

A *binary decision tree* (BDT) is a tree $t : \mathbb{N}^* \rightharpoonup \Sigma_\mathbb{X}$ such that the following two conditions hold (on top of the ones stated in Section 4.1):

1. $\forall p \in \mathrm{dom}(t) : t(p).var = x_i \wedge p' \in \{p.j | j \in \mathbb{N}, p.j \in \mathrm{dom}(t)\} \Rightarrow t(p').var = x_{i+1}$
2. $t(\varepsilon).var = x_i \Rightarrow \forall p \in \mathrm{dom}(t) : \#(t(p)) = 0 \Rightarrow |p| = n - i + 1$




(a) Box $\mathtt{L}_0$ (b) Box $\mathtt{L}_1$ (c) Box $\mathtt{H}_0$ (d) Box $\mathtt{H}_1$




(e) Box $\mathtt{X}$ (f) Box $\mathtt{L}_\oplus$ (g) Box $\mathtt{H}_\oplus$

Figure 4.1: Boxes used in ABDDs. The canonical form of an ABDD is obtained by using this box order: $(\mathtt{L_0}, \mathtt{H_0}, \mathtt{L_1}, \mathtt{H_1}, \mathtt{X}, \mathtt{L}_\oplus, \mathtt{H}_\oplus)$. Note that if in any box, both states indexed by 1 and 2 had self-loops, then that box would be nondeterministic (breaking condition 7 – unambiguity). All transitions in the boxes are short, and transitions labeled with symbol $\langle \mathtt{L}\colon \mathtt{S}, \mathtt{H}\colon \mathtt{S} \rangle$ are denoted using "hyperedges" with dashed and solid lines. For example, for a transition $q \text{–}\langle \mathtt{L}\colon \mathtt{S}, \mathtt{H}\colon \mathtt{S} \rangle\!\!\rightarrow (q_1, q_2)$, the dashed line symbolizes the $\mathtt{L}$ part of the transition (leading to $q_1$) and the solid line symbolizes the $\mathtt{H}$ part of the transition (leading to $q_2$).

The first condition ensures that consecutive nodes are labeled with consecutive variables. The second condition ensures that $t$ is a full binary tree with branches of the same length. The variable of a position $p$ is defined as $var(p) = x_{|p|+1}$. Every position $p \in \mathrm{dom}(t)$ of a BDT $t$ gives rise to the Boolean function $BF(t, p)$:

$$BF(t,p) = \begin{cases} t(p) & \text{if } t(p) \in \{\mathbf{0}, \mathbf{1}\} \\ (\neg t(p).var \wedge BF(t, p.0)) \vee (t(p).var \wedge BF(t, p.1)) & \text{otherwise.} \end{cases}$$

The Boolean function represented by the BDT $t$ is $BF(t, \epsilon)$.

## 4.3 ABDD canonization

Since all needed concepts have been defined, now a general overview of canonization algorithms for ABDDs will be presented. More detailed descriptions and definitions of algorithms are in [24].

An *unreduced binary decision automaton* (UBDA) is a TA $\mathcal{U} = \langle Q, \Sigma_{\mathtt{LH}\mathbb{X}}, \Delta, \mathcal{R}\rangle$. A *BDT run* of $\mathcal{U}$ on a BDT $t$ is a function $\rho\colon \mathrm{dom}(t) \to Q$, where:

$$\forall p \in \mathrm{dom}(t) : \rho(p) = q \wedge t(p) = \langle \mathtt{L}\colon \mathtt{S}, \mathtt{H}\colon \mathtt{S}, \mathit{var}\colon x\rangle \wedge \forall 0 \leq i \leq 1 : (q_i = \rho(t(p.i))) \implies$$
$$q \text{–}\boxed{\langle \mathtt{L}\colon \mathtt{S}, \mathtt{H}\colon \mathtt{S}\rangle}\!\to (q_0, \ldots, q_{n-1}) \in \Delta \vee q \text{–}\boxed{\langle \mathtt{L}\colon \mathtt{S}, \mathtt{H}\colon \mathtt{S}, \mathit{var}\colon x\rangle}\!\to (q_0, \ldots, q_{n-1}) \in \Delta.$$

Then, $t$ is in the *BDT language* of a state $q \in Q$ in $\mathcal{U}$, denoted as $t \in \mathcal{L}_{BDT}(\mathcal{U}, q)$ if there exists a BDT run $\rho$ of $\mathcal{U}$ on $t$ such that $\rho(\epsilon) = q$. The BDT language of $\mathcal{U}$ is $\mathcal{L}_{BDT}(\mathcal{U}) = \bigcup_{r\in\mathcal{R}} \mathcal{L}_{BDT}(\mathcal{U}, r)$. It represents all BDTs respecting the variables in the transitions of the UBDA.

$\mathcal{U}$ is called *well-specified* if its BDT language contains exactly one BDT $t$ and there exists exactly one run $\rho_t$ of $\mathcal{U}$ over $t$. The *semantics* of $\mathcal{U}$ is then defined as $\mathit{sem}(\mathcal{U}) = t$.

Let $\mathcal{U} = \langle Q, \Sigma_{\mathtt{LH}\mathbb{X}}, \Delta, \mathcal{R}\rangle$ be a well-specified UBDA such that $t = \mathit{sem}(\mathcal{U})$. The *run language* of a state $q \in Q$ is defined as $\mathcal{L}^\rho(q) = \{t' \mid \exists p \in \mathrm{dom}(t)\colon \rho_t(p) = q, t'$ is the subtree of $t$ at $p\}$; in other words, $\mathcal{L}^\rho(q)$ contains all subtrees of $t$ whose root is labeled by $q$ in $\rho_t$.

A well-specified UBDA $\mathcal{U}$ is *normalized*, if for all distinct states $q, q'$ of $\mathcal{U}$, it holds that $\mathcal{L}^\rho(q) \cap \mathcal{L}^\rho(q') = \emptyset$.

A *binary decision automaton* (BDA) over the set of boxes $\Gamma$ is a tree automaton $\mathcal{V} = \langle Q, \Sigma_{\mathtt{LH}\Gamma}, \Delta, \{r_\mathcal{V}\}\rangle$ such that if there is a transition $q \text{–}\boxed{\langle \mathit{in}\colon a\rangle_{/k}}\!\to (q_1, \ldots, q_k) \in \Delta$, then $q = r_\mathcal{V}$ and there is no other transition leaving $r_\mathcal{V}$. Well-specified and normalized BDAs are defined analogously to UBDAs.

The *unfolding* procedure traverses the UBDAs structure in a breadth-first traversal respecting the lexicographic order (low targets first, then high targets). During the traversal, every occurrence of a box in a transition is replaced with the transitions from that box, while correctly replacing all states with port transitions with the states from the initial transition.

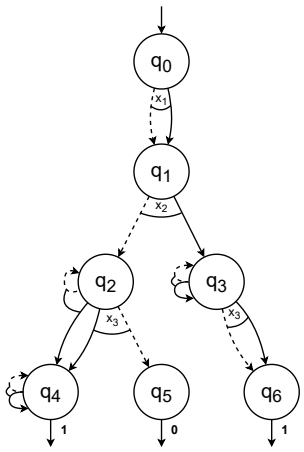
*Normalization* of a UBDA uses a bottom-up traversal of the UBDA, keeping track of variables (starting from a *leaf level* pseudo-variable $x_{n+1}$, descending) and merging the states with the same low and high descendants and the same variable.

The *folding* procedure takes a normalized UBDA, and based on the given order of the boxes, traverses the states of the UBDA structure in a lexicographic order (depth-first, low before high) and tries to "cover" a part of the UBDA with the given box. To achieve that, an intersection-like construction between the UBDA and the box is created, which tries to place the port transitions from the box to the UBDA, marking the UBDA's states as targets of a newly folded transition, along with variables, with which these states are accessed. This process can sometimes "disconnect" some states of the UBDA, making them top-down inaccessible. The more states are disconnected in such a way, the better the reduction. Using different sets of boxes in different orders can be used to simulate different BDD models.

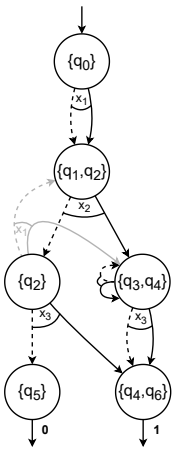
To obtain a canonical representation of a Boolean function using an automaton binary decision diagram $\mathcal{D}$ over the set of boxes $\Gamma$, the following sequence of operations is performed:

1. *unfold* boxes in $\mathcal{D}$ to obtain an unreduced binary decision automaton (UBDA) $\mathcal{U}_\mathcal{D}$,
2. obtain a *normalized* UBDA $\mathcal{U}'_\mathcal{D}$, and
3. *fold* $\mathcal{U}'_\mathcal{D}$ using the boxes from $\Gamma$ in a specified order.

An *automaton binary decision diagram* (ABDD) over boxes $\Gamma$ is defined as a BDA $\mathcal{D} = \langle Q, \Sigma_\Gamma, \Delta, \{r_\mathcal{D}\}\rangle$, such that:




(a) Normalization input




(b) Normalization output

Figure 4.2: Demonstration of an incorrect transition created by the previous normalization. The grey transition in (b) creates infeasible paths in the normalized UBDA and will not be created in the improved version.

1. $\mathcal{D}$ contains no loops,
2. $\mathcal{D}$ has no useless states,
3. every state $q \in Q$ has exactly one outgoing transition,
4. $\forall q \xrightarrow{a} (q_1, \ldots, q_k) \in \Delta : \forall 1 \leq i \leq k : var(q) < var(q_i) \vee var(q_i) = \bot$ (all paths in the ABDD respect the variable order), and
5. $\forall q \in Q : var(q) = \bot \iff \exists q \xrightarrow{a} () \in \Delta : a \in \{\mathbf{0}, \mathbf{1}\}$ (all states except leaves have to be labeled with a variable).

## 4.4 Further remarks

An improvement to the normalization algorithm used during canonization, presented in [24], is proposed here, see Algorithm 2. Some transitions in the normalized UBDA do not make sense with regards to the variable order, i.e. have no semantics within the result. Building of "intersectoids" during the folding procedure will still analyze these transitions. However, since they do not respect the variable order $\mathbb{X}$ (either they create paths that repeat a variable or even go against the order), they are ignored and take no part in creating the mapping for the ports of the box used as a reduction in the folded result.

The proposal uses a cache that stores the used variable for each created state set (or a macrostate of the resulting normalized UBDA). This cache (labeled $\beta$ in Algorithm 2) will allow the algorithm to check every created transition. If some transition's target state should have a smaller or equal cached variable than the source state, the transition is not created (line 11). The proposed change should lead to faster folding procedure, since less transitions have to be processed during creation of an intersectoid, and finding a mapping should also be faster. For demonstration of edges that have no semantics in the normalized UBDA, see Figure 4.2.

**Algorithm 2:** New UBDA normalization

**Input:** A well-specified UBDA $\mathcal{U} = \langle Q, \Sigma_{\mathtt{LH}\mathbb{X}}, \Delta, \mathcal{R}\rangle$, a variable order $\mathbb{X} = \langle x_1, x_2, \ldots, x_{n-1}, x_n\rangle$

**Result:** A normalized UBDA $\mathcal{U}'$ s.t. $\mathcal{L}_{BDT}(\mathcal{U}') = \mathcal{L}_{BDT}(\mathcal{U})$

$I_{\mathbf{0}} \leftarrow \{q \mid q \text{–}\boxed{\mathbf{0}}\!\!\rightarrow() \in \Delta\}$; $\quad I_{\mathbf{1}} \leftarrow \{q \mid q \text{–}\boxed{\mathbf{1}}\!\!\rightarrow() \in \Delta\}$;

$\Delta' \leftarrow \{I_{\mathbf{0}} \text{–}\boxed{\mathbf{0}}\!\!\rightarrow(), I_{\mathbf{1}} \text{–}\boxed{\mathbf{1}}\!\!\rightarrow()\}$; $\quad \mathcal{Q}' \leftarrow \{I_{\mathbf{0}}, I_{\mathbf{1}}\}$; $\quad \mathcal{R}' \leftarrow \emptyset$;

$\alpha \colon \mathbb{N} \rightharpoonup 2^{2^Q}$; $\quad \alpha[n+1] \leftarrow \{I_{\mathbf{0}}, I_{\mathbf{1}}\}$;

$\beta \colon 2^Q \rightharpoonup \mathbb{N}$; $\quad \beta[I_{\mathbf{0}}] \leftarrow n+1$; $\quad \beta[I_{\mathbf{1}}] \leftarrow n+1$;

**foreach** $x_i \in \langle x_{n+1}, x_n, x_{n-1}, \ldots, x_2, x_1\rangle$ **do**

  **foreach** $(U, V) \in \alpha[i] \times \alpha[i]$ **do**

    $W \leftarrow \{q \mid q \text{–}\boxed{\langle \mathtt{L}\colon \mathtt{S}, \mathtt{H}\colon \mathtt{S}, \mathit{var}\colon x_i\rangle}\!\!\rightarrow(u, v) \in \Delta \mid u \in U \wedge v \in V\} \cup$

      $\{q \mid q \text{–}\boxed{\langle \mathtt{L}\colon \mathtt{S}, \mathtt{H}\colon \mathtt{S}\rangle}\!\!\rightarrow(u, v) \in \Delta \mid u \in U \wedge v \in V\}$;

    **if** $W = \emptyset$ **then continue**;

    $\alpha[i-1] \leftarrow \alpha[i-1] \cup \{W\}$

    **if** $\beta[W] = \bot \vee (\beta[U] > \beta[W] \wedge \beta[V] > \beta[W])$ **then**

      $\beta[W] \leftarrow i$;

      **if** $W \in \{U, V\}$ **then** $\Delta' \leftarrow \Delta' \cup \{W \text{–}\boxed{\langle \mathtt{L}\colon \mathtt{S}, \mathtt{H}\colon \mathtt{S}\rangle}\!\!\rightarrow(U, V)\}$ ;

        **else** $\Delta' \leftarrow \Delta' \cup \{W \text{–}\boxed{\langle \mathtt{L}\colon \mathtt{S}, \mathtt{H}\colon \mathtt{S}, \mathit{var}\colon x\rangle_i}\!\!\rightarrow(U, V)\}$ ;

      $Q' \leftarrow Q' \cup \{W\}$;

      **if** $W \cap \mathcal{R} \neq \emptyset$ **then** $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{W\}$;

  $\beta \leftarrow \beta \cup \{q \mapsto i \mid q \in \alpha[i-1]\}$;

**return** $\mathcal{U}' = \langle \mathcal{Q}', \Sigma_{\mathtt{LH}\mathbb{X}}, \Delta', \mathcal{R}'\rangle$

# Chapter 5

# ABDD Apply

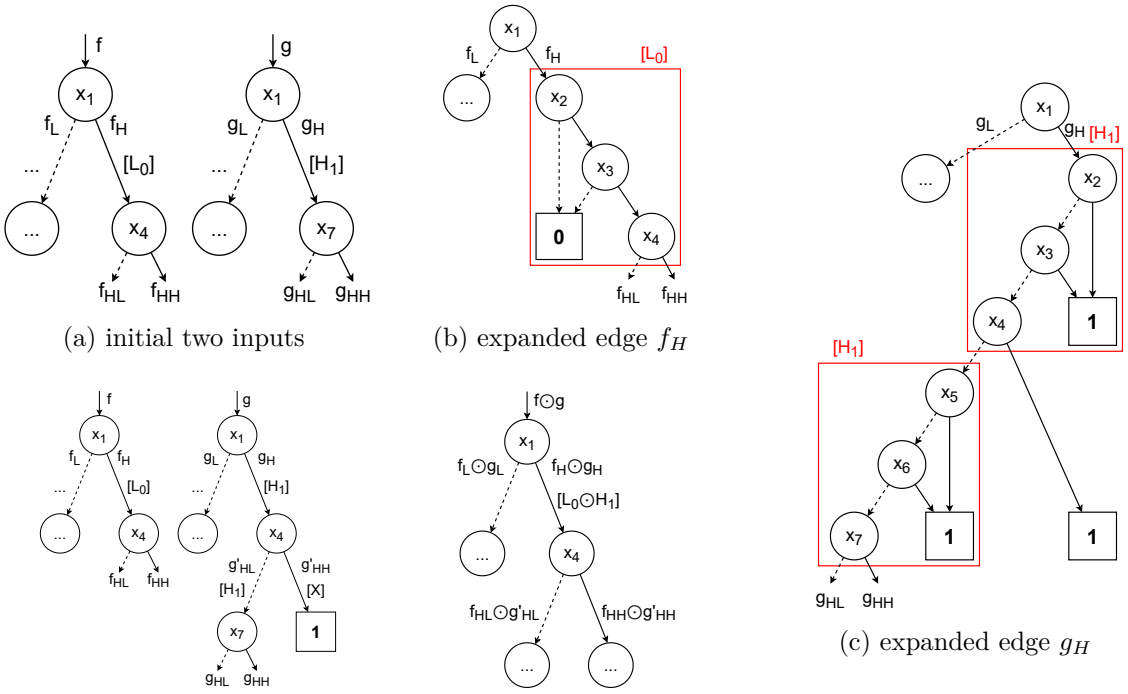
The following chapter explains all different techniques and constructions that were used to implement the Apply algorithm on ABDDs. Firstly, explanation of how different boxes can be applied together without unfolding is provided. Secondly, node materialization, i.e. a way to synchronize recursive descent within two different ABDD structures, such that node variables are the same, is explained. After that, additional optimizations that were utilized in the algorithm (memoization, etc.) are discussed. Finally, an overview of the whole algorithm is given.

It is important to have some intuition of why concepts like reduction rule merging and node materialization are important. A Boolean function in ABDDs is represented by an edge, which consists of the reduction rule and the target nodes (an edge can have multiple targets in case of boxes with multiple ports). Since the main requirement for the implementation of ABDD Apply is that the complexity of the algorithm depends on the sizes of the input graphs and not on the number of variables, the algorithm needs to be able to combine two edges (which can skip $n$ variables) in constant time.

Combining (or merging) two edges using a Boolean operator during Apply can only be done when the edges represent Boolean functions over the same domains. This means that the edges have to originate from nodes of the same variable (additionally, for correct edge combining, it is also enforced that the target nodes of both edges also share the same variable).

However, during Apply's traversal of two different ABDDs, where different reductions are used in different places (skipping different ranges of variables), the conditions for successful edge merging are not always met. For this reason, Apply will "reintroduce" (or arbitrarily create) an intermediate node on some edge, such that after this node is created, the conditions for edge merging are met and the recursive algorithm can proceed. In the worst case, every edge of the ABDD will require these arbitrary nodes, which will still keep the worst case complexity linear with regards to the input graph sizes. Notably, introducing a node on a reduced edge of an ABDD needs to be done with regards to the semantics of the box representing the reduction. A demonstration of one step of Apply on ABDDs for top-level understanding and intuition is in Figure 5.1.

Before explaining the Apply algorithm on ABDDs, some supporting notation is introduced, which is more natural to use and more closely reflects the implementation. Given a transition $\tau = q \text{-} \langle \mathtt{L}\colon \mathcal{B}_1, \mathtt{H}\colon \mathcal{B}_2, var\colon x_i \rangle \rightarrow (q_1, \ldots, q_a, q_{a+1}, \ldots q_{a+b}) \in \Delta$ in a BDA $\mathcal{D} = \langle Q, \Sigma_\Gamma, \Delta, \{r_\mathcal{D}\} \rangle$, such that $a = \#(\mathcal{B}_1), b = \#(\mathcal{B}_2)$ (assuming $\#(\mathtt{S}) = 1$), node $n = node\_cast(q)$ of an ABDD is obtained from $q \in Q$ in the following manner:




(a) initial two inputs (b) expanded edge $f_H$ (c) expanded edge $g_H$

(d) after materialization (e) output

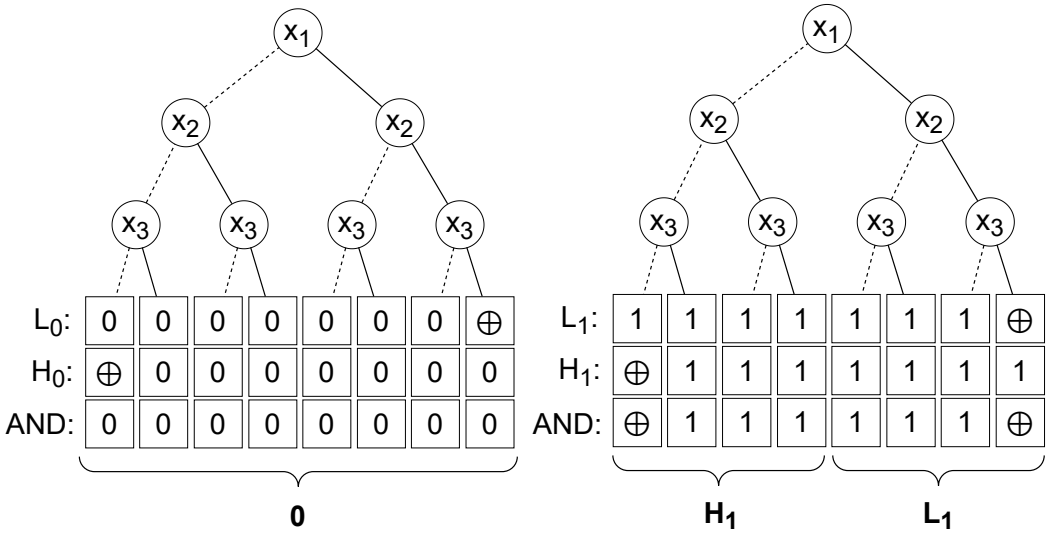
Figure 5.1: An overview of combining a pair of ABDD edges representing functions $f_H, g_H$ with operator $\odot$ – one call of the Apply algorithm. (a) shows the initial two ABDD structures surrounding edges $f_H, g_H$. (b) shows the "unfolded" version of $f_H$. (c) shows the "unfolded" version of $g_H$. The unfolded edges should provide intuition for why node materialization is necessary in Apply and how it should work in ABDDs. Since edge $g_H$ needs materialization, (d) shows how the structure of the second input is changed. Finally, (e) shows how the result is obtained. The resulting edge will use a reduction that is the result of merging the initial reduction rules (as denoted in the bracket $[\mathtt{L_0} \odot \mathtt{H_1}]$).

- $n.var = x_i$
- $n.lowbox = \mathcal{B}_1$
- $n.highbox = \mathcal{B}_2$
- $n.low = \langle node\_cast(q_1), \ldots, node\_cast(q_a)\rangle$,
- $n.high = \langle node\_cast(q_{a+1}), \ldots, node\_cast(q_{a+b})\rangle$
- $n.leaf = \bot$

For transitions $\kappa = q \text{–}\boxed{a}\!\rightarrow()$ where $a \in \{\mathbf{0}, \mathbf{1}\}$, all properties of $n = node\_cast(q)$ are $\bot$, except for $n.leaf = a$. In the following, an edge will refer to the pair $\langle n.lowbox, n.low\rangle$ or $\langle n.highbox, n.high\rangle$. For an edge $e$, $e.rule$ and $e.targets$ refers to the box and target nodes, respectively. The notion of an edge in this case is the same as the notion of a cofactor, as used in [3, 8]. The full ABDD can then be understood as a recursive data structure, represented by a root edge (a box with its target nodes), and each node (except leaves) is associated with a variable and low and high edges. Additionally, since within boxes, there are no box occurrences on edges $\text{–}\boxed{\langle \mathtt{L}\colon \mathtt{S}, \mathtt{H}\colon \mathtt{S}\rangle}\!\rightarrow$, a shortened notation $\text{–}\boxed{\mathtt{LH}}\!\rightarrow$ can be used.

## 5.1 Applying Boolean operations on reduced edges

In this section, combining edges using Boolean operators is explained. In each call of the Apply procedure, when processing two nodes $n_1, n_2$ from separate ABDDs, the procedure tries to combine the *low* cofactors of $n_1, n_2$, and then tries to combine the *high* cofactors of $n_1, n_2$, obtaining new cofactors $low_c, high_c$, which are then merged (and possibly, reduced) into the resulting node. Combining the semantics of the boxes using some Boolean operator can lead to a potentially predetermined result (short-circuiting), or it can lead to obtaining some other box's behavior. Doing this efficiently could avoid recursing to further calls of the Apply function. Given two edges within the ABDD, starting from the same variable level $x_i$, and leading to nodes of the same variable level $x_j$ (so, essentially, representing binary trees of equal height), one can easily perform Boolean operations on the leaf nodes' values of these trees, as demonstrated in Figure 5.2. The following text will explain how to perform this procedure on boxes instead of trees.




(a) The two reduction rules cancel out

(b) The result is more granular

Figure 5.2: The motivation behind combining boxes with Boolean operators. Two examples of using *AND* to evaluate the result of two aligned binary decision trees (obtained by unwinding the box reduction). (a) leads to short circuiting the evaluation to zero. (b) cannot be represented by a single box, but by introducing a root node, its two subtrees can be represented by the reduction rules of ABDDs.

Given two boxes $\mathcal{B}_1 = \langle Q_1, \Sigma_\oplus, \Delta_1, \mathcal{R}_1 \rangle$, $\mathcal{B}_2 = \langle Q_2, \Sigma_\oplus, \Delta_2, \mathcal{R}_2 \rangle$, and a binary Boolean operator $\odot$, the $\odot$-*product* of $\mathcal{B}_1, \mathcal{B}_2$ is a box $\mathcal{B} = \langle Q_1 \times Q_2, \Sigma_\oplus, \Delta_{\texttt{LH}} \cup \Delta_{out}, \mathcal{R}_1 \times \mathcal{R}_2 \rangle$, where:

$$\Delta_{\texttt{LH}} = \{\langle q, q' \rangle \text{—}\boxed{\texttt{LH}}\!\!\rightarrow (\langle q_1, q_1' \rangle, \langle q_2, q_2' \rangle) \mid q \text{—}\boxed{\texttt{LH}}\!\!\rightarrow (q_1, q_2) \in \Delta_1, q' \text{—}\boxed{\texttt{LH}}\!\!\rightarrow (q_1', q_2') \in \Delta_2\}$$

$$\Delta_{out} = \{\langle q, q' \rangle \text{—}\boxed{a}\!\!\rightarrow () \mid q \text{—}\boxed{a_1}\!\!\rightarrow () \in \Delta_1, q \text{—}\boxed{a_2}\!\!\rightarrow () \in \Delta_2, a \leftarrow op\_table[\odot, a_1, a_2].sym\}$$

$$f : Q \rightharpoonup Info = \{\langle q, q' \rangle \mapsto p \mid \langle q, q' \rangle \text{—}\boxed{a}\!\!\rightarrow () \in \Delta_{out}, p \leftarrow op\_table[\odot, a_1, a_2].info\}$$

Examples of *op_tables* for some basic Boolean operators are found in Table 5.1. The $\odot$-*product* describes what boxes will be used, but does not explain the semantics behind the ports. Each port from the initial boxes $\mathcal{B}_1, \mathcal{B}_2$ is mapped to a node from their respective

Table 5.1: Output transition rules for creating the $\odot$-*product* between boxes for 4 sample Boolean functions; the $\perp$ symbol means no meta-information needed, $i$ or $j$ without negation symbol $\neg$ refer to the case where the port-mapped node is used directly, with the negation symbol, the subgraph rooted in the port-mapped node needs to be negated, and $(i, j)$ means that the resulting node needs to be computed using apply on nodes port-mapped to by ports $\oplus_i, \oplus_j$. Note how the annihilation properties of AND and OR functions are utilized.

| AND | **0** | **1** | $\oplus_j$ |
|---|---|---|---|
| **0** | $(\mathbf{0}, \perp)$ | $(\mathbf{0}, \perp)$ | $(\mathbf{0}, \perp)$ |
| **1** | $(\mathbf{0}, \perp)$ | $(\mathbf{1}, \perp)$ | $(\oplus, j)$ |
| $\oplus_i$ | $(\mathbf{0}, \perp)$ | $(\oplus, i)$ | $(\oplus, (i, j))$ |

| OR | **0** | **1** | $\oplus_j$ |
|---|---|---|---|
| **0** | $(\mathbf{0}, \perp)$ | $(\mathbf{1}, \perp)$ | $(\oplus, j)$ |
| **1** | $(\mathbf{1}, \perp)$ | $(\mathbf{1}, \perp)$ | $(\mathbf{1}, \perp)$ |
| $\oplus_i$ | $(\oplus, i)$ | $(\mathbf{1}, \perp)$ | $(\oplus, (i, j))$ |

| XOR | **0** | **1** | $\oplus_j$ |
|---|---|---|---|
| **0** | $(\mathbf{0}, \perp)$ | $(\mathbf{1}, \perp)$ | $(\oplus, j)$ |
| **1** | $(\mathbf{1}, \perp)$ | $(\mathbf{0}, \perp)$ | $(\oplus, j, \neg)$ |
| $\oplus_i$ | $(\oplus, i)$ | $(\oplus, i, \neg)$ | $(\oplus, (i, j))$ |

| NAND | **0** | **1** | $\oplus_j$ |
|---|---|---|---|
| **0** | $(\mathbf{1}, \perp)$ | $(\mathbf{1}, \perp)$ | $(\mathbf{1}, \perp)$ |
| **1** | $(\mathbf{1}, \perp)$ | $(\mathbf{0}, \perp)$ | $(\oplus, j, \neg)$ |
| $\oplus_i$ | $(\mathbf{1}, \perp)$ | $(\oplus, i, \neg)$ | $(\oplus, (i, j))$ |

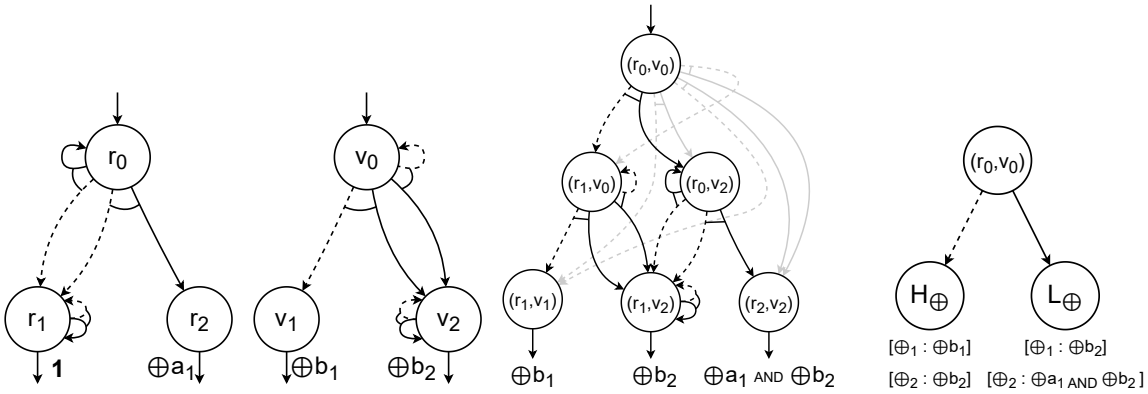
ABDDs. Based on the output symbols and the semantics of the $\odot$ operator, there are three situations that can happen with regards to the ports:

1. no recursion – the port-mapped node $n$ is used directly (from one of the initial edges),
2. negation – the port-mapped node $n$ is used in the negated form (so a negation of the subgraph rooted in $n$),
3. recursive Apply with $\odot$ – the port-mapped node is the result of a recursive apply call on nodes $n_1, n_2$.

The partial mapping $f$ maps the port-states of the $\odot$-*product* $\mathcal{B}$ to the necessary meta-information about the port. The meta information about the port says which ports from which boxes are used (which can be then directly mapped to the respective nodes of the initial ABDDs), if the resulting node mapped to the port should be negated, or if the resulting node mapped to the port is computed with a recursive apply call. Each element of *Info* can be formally defined as a tuple $(q_1, q_2, neg, rec)$, where:

- $q_1 \in Q_1 \cup \{\perp\}$, $q_2 \in Q_2 \cup \{\perp\}$ denote the port states from the first and second box respectively (or $\perp$ if unused).
- $neg \in \{0, 1\}$ denotes the negation flag. If $neg = 1$, then exactly one of $q_1, q_2$ must be some port state (and thus not $\perp$). If the flag is set, then the port-mapped node (based on the mapping $f$), that is the node of the ABDD which the port state represents, should be negated.
- $rec \in \{0, 1\}$ denotes the recursion flag. If $rec = 1$, both $q_1$ and $q_2$ must be port states (and not $\perp$) and the node mapped to the given port (based on the mapping $f$) is the result of the recursive Apply call of the corresponding nodes port-mapped to states $q_1, q_2$.

Just obtaining the $\odot$-*product* is not enough. In some cases (as demonstrated in Figure 5.3), the obtained automaton is not comparable to any of the used boxes. In that case, a recursive search of boxes rooted in states of the $\odot$-*product* is needed. During this search,




(a) Box $\mathtt{L_1}$ (b) Box $\mathtt{H}_\oplus$ (c) $\odot$-*product* of boxes $\mathtt{L_1}, \mathtt{H}_\oplus$. (d) Obtained box tree.

Figure 5.3: Demonstration of obtaining a $\odot$-*product*. Inputs are boxes $\mathtt{L_1}$ and $\mathtt{H}_\oplus$, and $\odot = AND$. Given the initial edges $e_1 = \langle \mathtt{L_1}, \langle a_1 \rangle\rangle, e_2 = \langle \mathtt{H}_\oplus, \langle b_1, b_2 \rangle\rangle$ going from nodes labeled with variable $x_i$, the obtained box tree can be used to get the edge $\langle \mathtt{S}, \langle n \rangle\rangle$, where $n$ is a newly created node, such that $n.var = next(x_i)$, $low(n) = \langle \mathtt{H}_\oplus, \langle b_1, b_2 \rangle\rangle$, $high(n) = \langle \mathtt{L}_\oplus, \langle b_2, c \rangle\rangle$, where $c = a_1 \odot b_2$. The greyed-out transitions were irrelevant to the box tree creation (i.e. using these lead to unsuccessful search of comparable boxes).

starting from the root state of the box, the automaton rooted in the visited state is compared with all of the boxes for language equality. If no box is equal, the search splits based on the outgoing $\xrightarrow{\langle \mathtt{L}\colon \mathtt{S}, \mathtt{H}\colon \mathtt{S} \rangle}$ transition. In case more outgoing transitions like that are present, each one of them is tried until one leads to a full box tree.

Experimenting with box $\odot$-*products*, and subsequently with the created box trees showed the following. Every possible pair of boxes from this model (see Figure 4.1), when combined using any of the binary Boolean operators, yields either a box tree with one leaf node (meaning it can be translated directly to some box), or a box tree with one inner node and two leaf nodes. This is promising, because it means that aside from the occasional need to introduce one node, the set of boxes is "closed" under the Boolean operators, meaning that there is no need to introduce a new box just to represent the semantics of some box product. There are two cases in which some special handling is needed. If a box product yields trivial false, then the resulting box would be $\mathtt{X}$ leading to a terminal $\mathbf{0}$ (similarly with trivial true). For example, if performing $\mathtt{L_0}$ *AND* $\mathtt{H_0}$, the result will be a box that has only $\mathbf{0}$-labeled output transitions. Notably, each leaf node $b$ of the box tree has the name of the box used ($b.box$), and the list of information ($b.info$) about how to map the ports (i.e. which nodes from the initial edge targets to use, whether to negate them, and if recursion is needed). In case the node is non-leaf, there is no information. Note that during Apply, when the box tree is traversed and some node has a set recursion flag, this can lead to indirect recursion (Apply → box tree processing → Apply ...) The procedure of obtaining the box product and the box tree is demonstrated in Figure 5.3.

## 5.2 Synchronizing variables during the recursive descent

The simultaneous recursive descent in both ABDD inputs needs to be done in such a way, that when cofactors are computed, the source node variables are equal. Shannon expansion [31], which describes how variables can arbitrarily be introduced into the existing

Boolean function, allows to guide the recursion. This becomes challenging, however, when a node with a specific variable has to be introduced on an already reduced edge, such that the box's semantics are kept intact. This process will be called *node materialization*.

Before going into the specifics of node materialization, some supporting terms have to be defined. Given some box $\mathcal{B}$, variables $x_{in}$ and $x_{out_i}$ for $\oplus_i \in \Sigma_\oplus$, a state $q$ of a box can "see" some variable $x_i$, if $\exists t \in \mathcal{L}(\mathcal{B}) : \exists p \in \mathrm{dom}(t) : \rho(p) = q \wedge t(p) = \langle \mathtt{L}\colon \mathtt{S}, \mathtt{H}\colon \mathtt{S}, var\colon x_i \rangle$. Essentially a state $q$ can see the variable $x_i$, if the box's run on this tree assigns $q$ to node labeled with $\langle \mathtt{L}\colon \mathtt{S}, \mathtt{H}\colon \mathtt{S}, var\colon x_i \rangle$. Notably, the input variable $x_{in}$ is the one assigned to the parent node of the ABDD, and the boxes on the *low* and *high* edges will start their run with the $next(x_{in})$ variable, where $next()$ refers to the following variable in the $\mathbb{X}$ ordering.

First, node materialization is explained in steps, and then each step will be elaborated on. Before the process is started, here is the information that is known beforehand and is relevant during the process: an edge of the ABDD $e = \langle \mathcal{B}, \langle out_1, \ldots out_{\#(\mathcal{B})} \rangle \rangle$ (it does not matter if the edge is low or high), a variable $x_{in}$, from which the edge starts, variables $x_{out_i}$ for all targets of the edge, and the materialized variable $x_m$.

1. First, the variable ranges for each state of box $\mathcal{B}$ are computed (see Algorithm 3).
2. Then, a materialized box $\mathcal{B}_m$ is created from $\mathcal{B}$ utilizing the variable range information. The materialized box transition graph can be thought of as consisting of two partitions (cut by transitions labeled $x_m$), the first partition consists of states that can see $x_m$, or can only see variables smaller than $x_m$. The second partition consists of states that can see variables larger than $x_m$.
3. Within $\mathcal{B}_m$, a two-phase search of boxes that can contain the semantics of the pre-materialized and post-materialized partitions is performed.
4. This will create a materialized "pattern", which can then be inserted into the ABDD.

Algorithm 3 describes how for each state of the box, the range of variables is computed. The algorithm utilizes information about looping transitions and terminable transitions (i.e. transitions whose children all have some output transition). The algorithm works by propagating the variable bounds top-down and bottom-up, until every state has both of the variable ranges bounds set. Since unambiguity and other properties of boxes are assumed, every bound for every state should be computed. After variable ranges are computed, the process of obtaining a materialized box follows.

In the following text, initial states refer to those that are in the domain of $\sigma$, i.e. have not been split or have been split and their variable range ends with the materialized variable. The target states refer to those that are in the domain of $\tau$, i.e. have been split and their ranges do not contain materialized variable $x_m$ (alternatively, domain of $\tau$ has states that can see $next(x_m)$). Given a box $\mathcal{B} = \langle Q, \Sigma_\oplus, \Delta, \mathcal{R} \rangle$:

- $L = \{q \text{—}\boxed{\mathtt{LH}}\!\!\rightarrow (q_1, q_2) \in \Delta : \forall t \in \{q_1, q_2\} : t \in reachable(\langle Q, \Sigma, \Delta, \{t\} \rangle)\}$ is the set of self-looping transitions,
- $T = \{q \text{—}\boxed{\mathtt{LH}}\!\!\rightarrow (q_1, q_2) \in \Delta : \forall t \in \{q_1, q_2\} : t \notin reachable(\langle Q, \Sigma, \Delta, \{t\} \rangle)\} \cup \{q \text{—}\boxed{a}\!\!\rightarrow () \in \Delta : a \in \Sigma_\oplus\}$ is the set of terminating transitions (i.e. those that are used when the state has to stop looping),
- $\sigma : Q \rightharpoonup \mathbb{X} \times \mathbb{X} = \{q \mapsto [minv(q), (minv(q) \leq x_m < maxv(q) ? x_h : maxv(q))]\}$ contains information about variable ranges for all states from the initial box, however, for those states whose ranges did contain the materialized variable, it contains modified ranges, and

**Algorithm 3:** Compute variable ranges for box

**Input:** A box $\mathcal{B} = \langle Q, \Sigma_\oplus, \Delta, \{q_r\}\rangle$, an input variable $x_{in}$, output variable mapping $out : Q \rightharpoonup \mathbb{X}$

**Result:** Two mappings $minv, maxv : Q \to \mathbb{X}$, which define the variable ranges that each state within the box can "see"

$minv = \{q_r \mapsto next(x_{in})\} \cup \{q \mapsto \bot : q \in Q \setminus \{q_r\}\}$;
$maxv = \{q \mapsto out(q) : q \in \mathrm{dom}(out)\} \cup \{q \mapsto \bot : q \in Q \setminus \mathrm{dom}(out)\}$;
$T = \{q \text{–}\boxed{\texttt{LH}}\!\to (q_1, q_2) \in \Delta : a \in \Sigma_\oplus : \exists q_1 \text{–}\boxed{a}\!\to () \in \Delta, \exists q_2 \text{–}\boxed{a}\!\to () \in \Delta\}$;
$L = \{q \text{–}\boxed{\texttt{LH}}\!\to (q_1, q_2) \in \Delta : \forall t \in \{q_1, q_2\} : t \in reachable(\langle Q, \Sigma, \Delta, \{t\}\rangle)\}$;
**do**
  **foreach** $q \in Q$ **do**
    **if** $\nexists q \text{–}\boxed{\texttt{LH}}\!\to (q_1, q_2) \in L \wedge maxv(q) \neq \bot \wedge minv(q) = \bot$ **then** $minv(q) \leftarrow maxv(q)$;
    **if** $\nexists q \text{–}\boxed{\texttt{LH}}\!\to (q_1, q_2) \in L \wedge maxv(q) = \bot \wedge minv(q) \neq \bot$ **then** $maxv(q) \leftarrow minv(q)$;
    **foreach** $q \text{–}\boxed{\texttt{LH}}\!\to (q_1, q_2) \in T$ **do**
      **if** $\exists c \in \{t_1, t_2\} : minv(c) \neq \bot, t \text{–}\boxed{\texttt{LH}}\!\to (t_1, t_2) \in L, maxv(t) = \bot$ **then** $maxv(t) \leftarrow minv(c)$;
    **foreach** $l_i \in \{l_1, l_2\} : l \text{–}\boxed{\texttt{LH}}\!\to (l_1, l_2) \in L, minv(l) \neq \bot$ **do**
      **if** $l_i \neq l \wedge minv(l_i) = \bot$ **then** $minv(l_i) \leftarrow next(minv(l))$;
**while** *minv* or *maxv* get updated;
**return** $minv, maxv$

- $\tau : Q \rightharpoonup \mathbb{X} \times \mathbb{X} = \{q \mapsto [next(x_m), maxv(q)] : minv(q) \leq x_m < maxv(q)\}$ contains information about the states that had to be split due to their variable range containing materialized variable, these states are targets of transitions that contain the materialized variable in their symbol.

Utilizing the aforementioned constructs, for all $q \mapsto [x_l, x_h] \in \sigma$:

$$\Delta_1 = \{\langle q, x_l, x_h\rangle \text{–}\boxed{\texttt{LH}}\!\to (t_1, t_2) : x_l \neq x_h, q \text{–}\boxed{\texttt{LH}}\!\to (q_1, q_2) \in L \ \wedge \forall i \in \{1, 2\} : t_i = \langle q_i, x_l^{(i)}, x_h^{(i)}\rangle \wedge q_i \mapsto [x_l^{(i)}, x_h^{(i)}] \in \sigma\}$$

$$\Delta_2 = \{\langle q, x_l, x_h\rangle \text{–}\boxed{a}\!\to () : q \notin \mathrm{dom}(\tau), q \text{–}\boxed{a}\!\to () \in T\} \cup \{\langle q, x_l, x_h\rangle \text{–}\boxed{\texttt{LH}}\!\to (\alpha(q_1), \alpha(q_2)) : q \notin \mathrm{dom}(\tau), q \text{–}\boxed{\texttt{LH}}\!\to (q_1, q_2) \in T\},$$

$$\alpha(q_i) = \begin{cases} \langle q_i, x_l^{(i)}, x_h^{(i)}\rangle : q_i \mapsto [x_l^{(i)}, x_h^{(i)}] \in \tau & \text{if } q_i \in \mathrm{dom}(\tau) \wedge next(x_m) \in [x_l^{(i)}, x_h^{(i)}] \\ \langle q_i, x_l^{(i)}, x_h^{(i)}\rangle : q_i \mapsto [x_l^{(i)}, x_h^{(i)}] \in \sigma & \text{otherwise} \end{cases}$$

$$\Delta_3 = \{\langle q, x_l, x_h\rangle \text{–}\boxed{\texttt{LH}, var : x_m}\!\to (\beta(q_1), \beta(q_2)) : q \in \mathrm{dom}(\tau), q \text{–}\boxed{\texttt{LH}}\!\to (q_1, q_2) \in L\},$$

$$\beta(q_i) = \begin{cases} \langle q_i, x_l^{(i)}, x_h^{(i)}\rangle : q_i \mapsto [x_l^{(i)}, x_h^{(i)}] \in \tau & \text{if } q_i \in \mathrm{dom}(\tau) \\ \langle q_i, x_l^{(i)}, x_h^{(i)}\rangle : q_i \mapsto [x_l^{(i)}, x_h^{(i)}] \in \sigma & \text{otherwise} \end{cases}$$

$$\Delta_4 = \{\langle q, x_l, x_h\rangle \text{–}\boxed{a}\!\to () : a \in \{\mathbf{0}, \mathbf{1}\}, x_m \in [x_l, x_h], q \text{–}\boxed{a}\!\to () \in \Delta\} \cup \{\langle q, x_l, x_h\rangle \text{–}\boxed{\oplus_{arb}}\!\to () : x_m \in [x_l, x_h], q \text{–}\boxed{\oplus_i}\!\to () \in \Delta\}$$

$\Delta_1$ contains self-loops of the initial states. $\Delta_2$ consists of terminating transitions from the initial states which have no copy in target states. Transitions from $\Delta_3$ go from the original states to their copy in the target states, are created from the self-loops found in the original box, and are labeled with $x_m$. $\Delta_4$ is the set of arbitrary port transitions added to the source states of transitions labeled with variable $x_m$. Now for all $q \mapsto [x_l, x_h] \in \tau$ (targets of the transitions with materialized variable):

$$\Delta_5 = \{\langle q, x_l, x_h\rangle \text{–}\boxed{\texttt{LH}}\text{→}(\beta(q_1), \beta(q_2)) : x_l \neq x_h, q \text{–}\boxed{\texttt{LH}}\text{→}(q_1, q_2) \in L\}$$

$$\Delta_6 = \{\langle q, x_l, x_h\rangle \text{–}\boxed{a}\text{→}() : q \text{–}\boxed{a}\text{→}() \in T\} \cup$$
$$\{\langle q, x_l, x_h\rangle \text{–}\boxed{\texttt{LH}}\text{→}(\gamma(q_1), \gamma(q_2)) : q \text{–}\boxed{\texttt{LH}}\text{→}(q_1, q_2) \in T\}$$

$$\gamma(q_i) = \begin{cases} \langle q_i, x_l^{(i)}, x_h^{(i)}\rangle : q_i \mapsto [x_l^{(i)}, x_h^{(i)}] \in \tau & \text{if } q_i \in \mathrm{dom}(\tau) \wedge x_h < x_h^{(i)} \\ \langle q_i, x_l^{(i)}, x_h^{(i)}\rangle : q_i \mapsto [x_l^{(i)}, x_h^{(i)}] \in \sigma & \text{otherwise} \end{cases}$$

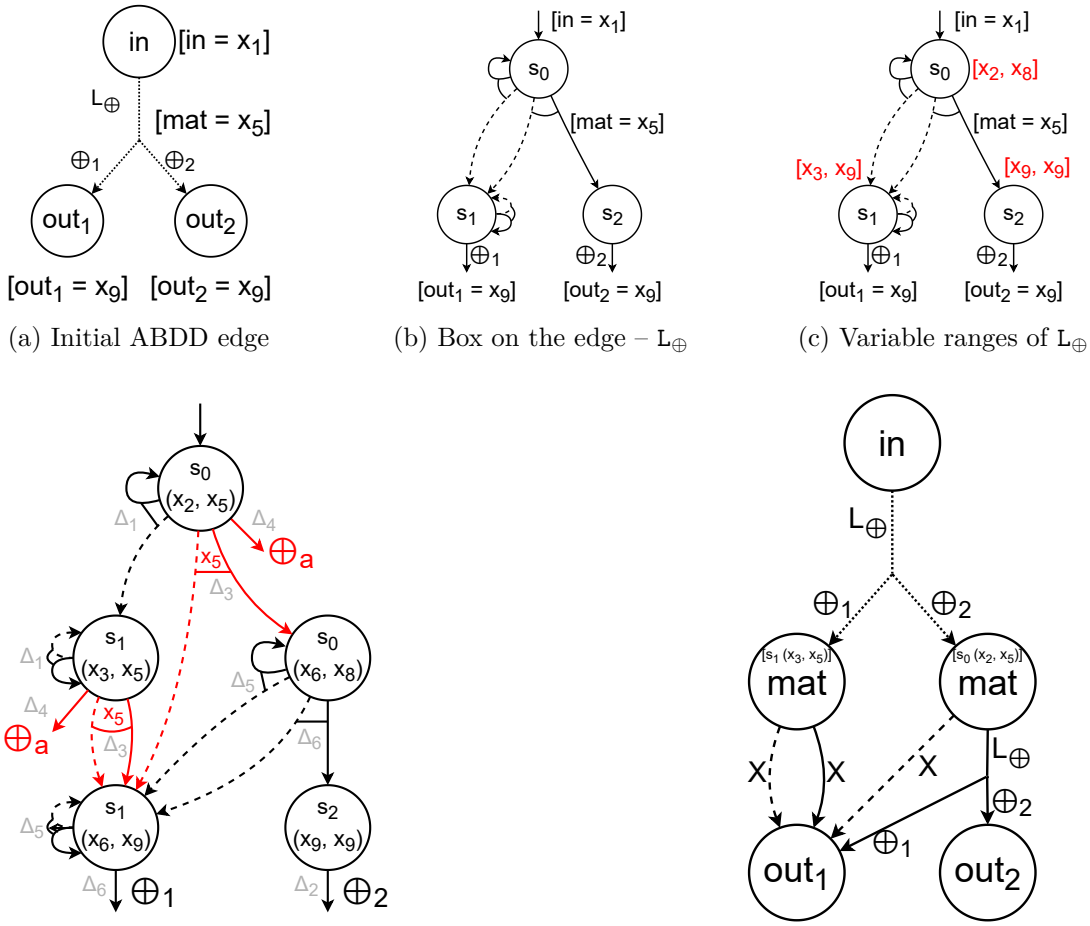
$\Delta_5$ contains copies of self-loops for the targets of the transitions with $x_m$. $\Delta_6$ is the set of terminating transitions of the target of the $x_m$-labeled transitions. Finally, the materialized box is $\mathcal{B}_m = \langle Q_m, \Sigma_m, \Delta_m, \mathcal{R}_m\rangle$, where:

- $Q_m = \{\langle q, x_l, x_h\rangle : q \mapsto [x_l, x_h] \in \tau \cup \sigma\}$,
- $\Sigma_m = \Sigma_\oplus \cup \{\oplus_{arb}\}$,
- $\Delta_m = \bigcup_{i=1}^{6} \Delta_i$,
- $\mathcal{R}_m = \{(q, x_l, x_h) : q \in \mathcal{R}, x_l = next(x_{in})\}$.

After the box is materialized with regards to variable $x_m$, a two-phase search of other boxes within it is started (which will also be referred to as "pattern finding"– finding the ABDD patterns within the materialized box).

1. First, the transitions with materialized variable (from $\Delta_3$) are removed.
2. If $\mathcal{L}(\mathcal{B}_m) \setminus T_\perp \subseteq \mathcal{L}(\mathcal{B})$, where $T_\perp = \{\{\varepsilon \mapsto \oplus_i\} \mid \oplus_i \in \Sigma_\oplus \cup \Sigma_m\}$ denotes the language of trivial trees (i.e. trees of only one node and the node is labeled with some port), and $\mathcal{B}$ is the box from the initial edge, then the box on the initial edge will stay as $\mathcal{B}$, otherwise `S`.
3. Then, before the second search, transitions from $\Delta_3$ are placed back and each `L` and `H` target state of the transition becomes the root of the box that is compared (wrt. languages) to other boxes. So for transitions labeled $\text{–}\boxed{\texttt{LH}, var : x_m}\text{→}$ leading $q_1, \ldots, q_n$, comparison of $\mathcal{B}_m$ rooted in $q_1, \ldots, q_n$ is performed. The comparison checks if $\mathcal{L}(\mathcal{B}_m) \setminus T_\perp \subseteq \mathcal{L}(\mathcal{B})$, and similarly to the first search, if such a box $\mathcal{B} \in \Gamma$ is found, it will label the `L` or `H` edge in the resulting ABDD.

The whole box materialization process, including variable ranges computation, is shown in Figure 5.4.




(a) Initial ABDD edge

(b) Box on the edge – $\mathtt{L}_{\oplus}$

(c) Variable ranges of $\mathtt{L}_{\oplus}$

(d) Box $\mathtt{L}_{\oplus}$ materialized at $x_5$

(e) Modified ABDD structure

Figure 5.4: Each step of the materialization process. Firstly, (a) shows the known information before the materialized process, i.e the initial box, and the input, output, materialized variables. Secondly, (b) and (c) show how this information is mapped to the states of the box and how variable ranges are computed. Most importantly, in (d), the materialized box (with regards to given variable information) is shown. Grey markings show the types of transitions obtained during the construction, red markings show the transitions important during the "pattern finding" step. Note how these red transitions align with the layer of edges above the $out_1$ and $out_2$ labeled nodes in (e).

## 5.3 Additional optimizations

Since computing $\odot$-products of boxes (with their related box trees) is a costly operation, it is undesirable to perform them during the execution of the Apply algorithm. Because there is a finite number of Boolean operators, and a finite number of boxes, each box tree can be precomputed and stored in some cache, ready to be instantly loaded when needed during the algorithm. Since there are 7 boxes, and currently supported are 7 binary Boolean operators (AND, OR, NAND, NOR, XOR, IFF, IMPLY), in total there are $7^3$ box trees to be computed. During Apply, when needed, the box tree cache is accessed using the two rules on edges and the operator, and then the box tree can be processed, altering the resulting ABDD structure.

Similarly, box materialization is also a demanding operation, which would hinder the performance of Apply on ABDDs. Inspecting the properties of each box revealed that there is also a limited number of possible patterns that can emerge from materialization (based on the relationships between $x_{in}, x_m, x_{out_i}$). Box $\mathtt{X}$ has only 4 such patterns, boxes $\mathtt{L_1}, \mathtt{L_0}, \mathtt{H_1}, \mathtt{H_0}$ have 6 different patterns, and boxes $\mathtt{L}_\oplus, \mathtt{H}_\oplus$ have 8 (so 44 in total).

For each different materialized ABDD pattern of a box $\mathcal{B}$, a different set of predicates holds. These predicates describe relationships between the following pairs of variables: $(x_{in}, x_m), (x_{out_i}, x_m)$ for $1 \leq i \leq \#(\mathcal{B})$, $(x_{leaf}, x_m)$, where ($x_{leaf}$ is the pseudo-variable describing the leaf level – it is $next(x_{max})$ if $x_{max}$ is the largest variable in the order $\mathbb{X}$. For the purposes of ABDDs, there are two relevant predicates comparing the variables:

- $x_1 <_1 x_2 \equiv x_1 + 1 = x_2$
- $x_1 \ll x_2 \equiv x_1 + 1 < x_2$

For the $\mathtt{L}_\oplus$ box from Figure 5.4, the following predicates hold: $x_{in} \ll x_m \wedge x_m \ll x_{out_1} \wedge x_m \ll x_{out_2}$. Note that there are only the $<_1$ and $\ll$ predicates used, and not their $>_1, \gg$ counterparts (so the left-hand side always features the smaller variable). This is to ensure uniqueness of the predicate set (i.e. that there is only one way to represent the relationships – this is important for hashing and cache lookup). If neither $<_1, \ll$ can hold with regards to some variable, then it is assumed that $\geq$ holds (and the set does not include $<_1$ or $\ll$ for that given relationship). The process of generating all possible sets of predicates for some box $\mathcal{B} \in \Gamma$ is the following:

1. Generate different variable assignments for $x_{in}, x_{out_i}, x_m$ (and $x_{leaf}$ if needed, i.e. for boxes $\mathtt{L_0}, \mathtt{L_1}, \mathtt{H_0}, \mathtt{H_1}$).
2. Check the "consistency" of this assignment. Inconsistent assignments either do not warrant materialization, or cannot happen in a valid ABDD structure. The following conditions have to hold:
   - $x_{in} < x_m$,
   - $\exists 1 \leq i \leq \#(\mathcal{B}) : x_{out_i} > x_m$,
   - $x_{leaf} > x_m$,
   - all $x_{out_i}$ are consistent with the semantics of the box $\mathcal{B}$ (for example, in $\mathtt{L}_\oplus$, it is inherently required that $x_{out_1} \geq x_{out_2}$, in $\mathtt{H}_\oplus$ this is reversed), and
   - $\forall x_j \in \{x_{in}, x_{out_i} : 1 \leq i \leq \#(\mathcal{B})\} : x_j < x_{leaf}$.
3. Create a set of predicates from the consistent assignment and if such a set has not yet been evaluated, compute a materialized box $\mathcal{B}_m$ with regards to the given assignment.
4. Create a mapping of the predicate set to the obtained ABDD pattern from $\mathcal{B}_m$. Note that the pattern only contains symbolic variable names and not their values from the assignment. The variables are replaced with their actual values during the traversal of the pattern, when needed.

During the call of the Apply algorithm, sometimes a node is created which is exactly the same as some other node already used (perhaps in one of the input ABDDs or has already been created by the algorithm). Creating a duplicate node would be undesirable, so for this reason a *node cache* is utilized during the whole algorithm (sometimes also called

*vertex table*, *unique table*). The index to this table has to take into account every attribute of the ABDD node that can be used to distinguish it from other nodes. Every node $n$ of the ABDD has an entry in this table that looks like this: $\langle v, l, \mathcal{B}_L, \tau_V \mathcal{B}_H, \tau_H \rangle \mapsto n$, where:

- $v \in \mathbb{X} \cup \{x_{n+1}\} = n.var$,
- $l \in \{0, 1, \bot\}$ is the leaf value of node $n$ ($\bot$ in case of inner nodes),
- $\mathcal{B}_L = n.lowbox \in \Gamma \cup \{\mathtt{S}\}$ is the box used on the low edge of the node,
- $\tau_L = n.low$ is the tuple $\langle n_1, \ldots, n_{\#(\mathcal{B}_L)} \rangle$ that are the target nodes of the low edge
- $\mathcal{B}_H = n.highbox$, and $\tau_H$ are defined analogously to $\mathcal{B}_L, \tau_L$.

Usually, during Apply, *dynamic programming* technique can be utilized in the form of *call cache*. The call cache stores results of different Apply calls in hopes that when Apply reaches the same pair of edges (which is possible since ABDD is a DAG), the result does not have to be computed again and can be directly returned. Call cache stores entries in the form $(\odot, v, e_1, e_2) \mapsto e_R$, where $\odot$ is a binary Boolean operator, $v \in \mathbb{X}$ is the variable of the node from which the apply on the two edges $e_1, e_2$ is called. $e_1, e_2, e_R$ are ABDD edges in the form $\langle \mathcal{B}, \langle n_1, \ldots, n_{\#(\mathcal{B})} \rangle \rangle$.

Because negation has a different arity than other supported Boolean operators, there is also a *negation cache* utilized in the algorithm. It has entries in the form $n \mapsto m$, where $n, m$ are ABDD nodes. Negation is implemented as a recursive traversal of the subtree rooted in node $n$ which just switches terminal nodes and negates boxes on edges ($\mathtt{L_1} \leftrightarrow \mathtt{L_0}, \mathtt{H_1} \leftrightarrow \mathtt{H_0}$), while checking newly created nodes against the node cache.

In some cases, an unnecessary node materialization or box tree traversal can be skipped, when one of the edges describes a trivial Boolean function (true or false). Knowing this, the evaluation from this point can be simplified. Either a trivial function is returned ($\langle \mathtt{X}, \mathbf{0} \rangle$ for false, $\langle \mathtt{X}, \mathbf{1} \rangle$ for true – short circuit evaluation), or one of the edges is returned directly, or in a negated form.

## 5.4 Complete algorithm

Since all important aspects of Apply have been explained, now a complete overview of the Apply algorithm on ABDDs can be presented. Algorithm 4 describes the recursive function Apply.

First, the call cache is checked, then a short-circuit evaluation is attempted. When the short-circuit evaluation cannot be performed (when no edge describes a trivial Boolean function), it returns an empty tuple of states. Otherwise the result of short-circuiting is cached and returned. Afterwards, node materialization happens (if needed). Firstly, the algorithm computes materialization variable $x_m$, which is the minimum variable of target nodes, and then, comparing it to the maximum variable of either edge targets, the algorithm attempts to materialize the given edge.

The function *materialize*() computes the set of predicates based on the variable in edge's source node $n_i$, variables in edge's target nodes $e_i.target$, and the materialized variable $x_m$. Then, based on the edge's box $e_i.rule$ and the obtained predicate set, it selects the precomputed ABDD pattern from the cache and by traversing the pattern, it creates the necessary edges and nodes (replacing the symbolic names of the nodes with correct variables, or leaf nodes). A new edge (which would stem from the node $n_i$) is created, which is then used in

**Algorithm 4:** *ABDD_apply_from* function

**Input:** Boolean binary operator $\odot$, variable $x_v \in \mathbb{X}$, two ABDD edges $e_1, e_2$, nodes $n_1, n_2$ from which edges $e_1, e_2$ start

**Result:** ABDD edge $e_R = \langle \mathcal{B}_R, \langle n_1, \ldots, n_k \rangle \rangle$, where $k = \#(\mathcal{B})$

**if** `call_cache`$[\odot, x_v, e_1, e_2] \mapsto e_R$ **then return** $e_R$;
$\mathcal{B}_R, \tau_R \leftarrow short\_circuit\_eval(\odot, e_1, e_2)$;
**if** $\tau_R \neq \langle\rangle$ **then**
    `call_cache`$[\odot, x_v, e_1, e_2] \leftarrow \langle \mathcal{B}_R, \tau_R \rangle$;
    **return** $\mathcal{B}_R, \tau_R$
$x_m = \min\{n.var : n \in e_1.targets \vee n \in e_2.targets\}$;
**if** $x_m \neq max\{n.var : n \in e_1.targets\}$ **then**
    $e_m = materialize(n_1, e_1, x_m)$;
    **return** $ABDD\_apply\_from(\odot, x_m, e_m, e_2, n_1, n_2)$
**if** $x_m \neq max\{n.var : n \in e_2.targets\}$ **then**
    $e_m = materialize(n_2, e_2, x_m)$;
    **return** $ABDD\_apply\_from(\odot, x_m, e_1, e_m, n_1, n_2)$
**if** $\forall n : n \in e_1.target \vee n \in e_2.target : n.leaf \neq \bot$ **then**
    $b =$ `box_tree_cache`$[\odot, e_1.rule, e_2.rule]$;
    $x_s = \min\{n_1.var, n_2.var\}$;
    $\mathcal{B}_R, \tau_R = process\_box\_tree\_leaves(b, \odot, x_m, e_1, e_2, x_s)$;
    `call_cache`$[\odot, x_m, e_1, e_2] \leftarrow \langle \mathcal{B}_R, \tau_R \rangle$;
    **return** $\mathcal{B}_R, \tau_R$
$b =$ `box_tree_cache`$[\odot, e_1.rule, e_2.rule]$;
$x_s = \min\{n_1.var, n_2.var\}$;
$\mathcal{B}_R, \tau_R = process\_box\_tree(b, \odot, x_m, e_1, e_2, x_s)$;
`call_cache`$[\odot, x_m, e_1, e_2] \leftarrow \langle \mathcal{B}_R, \tau_R \rangle$;
**return** $\mathcal{B}_R, \tau_R$

a subsequent Apply call. Note that materialization does not actually modify the structure of any of the input ABDDs, it merely simulates the creation of edges and nodes on the call stack.

After materialization is done, if both edge targets contain only leaves, a special way of processing box tree is called, *process_box_tree_leaves*(). Newly created nodes of a boxtree are labeled, starting from variable $x_s$. The procedure does not invoke any further Apply calls, simply combining the values of leaf nodes using $\odot$ Boolean operator.

If edge targets are not strictly leaves, a similar function is called, *process_box_tree*(). This function, when it encounters a set *recurse* flag in some box tree node, will call Apply on the nodes specified by $idx_1, idx_2$. Notably, function *process_box_tree*() (see Algorithm 5) uses two variables $x_s, x_v$. $x_s$ labels the variable of the source nodes and is used to label newly created nodes of the box tree. $x_v$ labels the target nodes (they should have the same variable thanks to node materialization) and are used for propagating through other Apply calls. Demonstration of *process_box_tree*() is in Figure 5.5.

One of the strengths of the ABDD's implementation of Apply is that based on the nature of reduction rules, performing box operations beforehand can reveal some interesting way of short-circuiting the result. For example, $\mathtt{L_0}$ AND $\mathtt{H_0}$ leads to $\mathbf{0}$. This case is not

**Algorithm 5:** *process_box_tree* function

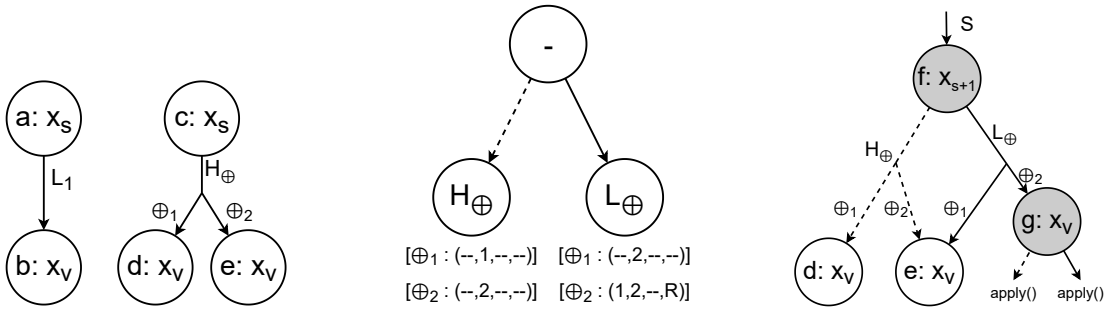
**Input:** node of a boxtree $b$, Boolean binary operator $\odot$, variable $x_t \in \mathbb{X}$ labeling target nodes, two ABDD edges $e_1, e_2$, variable $x_s \in \mathbb{X}$ labeling the source nodes of edges $e_1, e_2$

**Result:** ABDD edge $e_R = \langle \mathcal{B}_R, \langle r_1, \ldots, r_k \rangle \rangle$, where $k = \#(\mathcal{B})$

**if** *b is a leaf node* **then**
  **if** $b.box \in \{\mathbf{0}, \mathbf{1}\}$ **then return** $\langle \mathtt{X}, b.box \rangle$;
  $\mathcal{B} \leftarrow b.box$;
  **foreach** $pc_i \in b.info$ **do**
    **if** $pc_i.recursion$ **then**
      $n_1 = e_1.target[pc_i.idx_1], \quad n_2 = e_2.target[pc_i.idx_2]$;
      $e_{l_1} = \langle n_1.lowbox, n_1.low \rangle, \quad e_{l_2} = \langle n_2.lowbox, n_2.low \rangle$;
      $e_{h_1} = \langle n_1.highbox, n_1.high \rangle, \quad e_{h_2} = \langle n_2.highbox, n_2.high \rangle$;
      $\mathcal{B}_L, \tau_L = ABDD\_apply\_from(\odot, x_t, e_{l_1}, e_{l_2}, n_1, n_2)$;
      $\mathcal{B}_H, \tau_H = ABDD\_apply\_from(\odot, x_t, e_{h_1}, e_{h_2}, n_1, n_2)$;
      $r_i = \langle x_t, \mathcal{B}_L, \tau_L, \mathcal{B}_H, \tau_H \rangle$;
      node_cache.*check*$(r_i)$;
    **else if** $pc_i.idx_1 \neq \perp$ **then**
      $r_i = pc_i.negation \ ? \ negate(e_1.target[pc_i.idx_1]) \ : e_1.target[i.idx_1]$;
    **else if** $pc_i.idx_2 \neq \perp$ **then**
      $r_i = pc_i.negation \ ? \ negate(e_2.target[pc_i.idx_2]) \ : e_2.target[pc_i.idx_2]$;
  **return** $\langle \mathcal{B}, \langle r_1, \ldots r_{\#(\mathcal{B})} \rangle \rangle$
$\mathcal{B}_L, \tau_L = process\_box\_tree(b.low, \odot, x_t, e_1, e_2, next(x_s))$;
$\mathcal{B}_H, \tau_H = process\_box\_tree(b.high, \odot, x_t, e_1, e_2, next(x_s))$;
$r = \langle x_s, \mathcal{B}_L, \tau_L, \mathcal{B}_H, \tau_H \rangle$;
node_cache.*check*$(r)$;
**return** $\langle \mathtt{S}, \langle r \rangle \rangle$

detected instantly in other Apply implementations (that utilize both of these rules, such as ESRBDDs and CESRBDDs), where they are first expanded, then short-circuited, and during the phase of *cofactor merging* the resulting node is detected as reducible by some pattern, and perhaps after that the edge $\langle \mathtt{X}, \mathbf{0} \rangle$ is returned.

The main drawback of Apply on ABDDs is that the result is not canonical. Currently, there is no way to detect if a node is reducible by some pattern. For example, in BDDs, after low and high cofactors are created, a check is performed. If for node $n$ it holds that $low(n) = high(n)$, node $n$ is redundant, can be deleted, and either $low(n)$ or $high(n)$ is returned instead. In this way, the Apply procedure already ensures the result will be canonical. Current ABDD implementation does not possess such mechanism. During the algorithm, several reducible nodes can be created, and only after the initial recursive Apply call is done, the result will be reduced (using the canonization algorithms – unfolding, normalization, folding). There is no known systematic way of checking whether or not the newly created nodes or patterns are reducible. Even though for some reduction rules, the checking is simple (BDDs or ZBDDs for example), the problem becomes challenging if it needs to hold up even when adding new reduction rules, or even just changing the order of the box reductions, which is the feature of ABDDs. To ensure canonicity, the *cofactor merging* process should take into account which boxes are used, what is their reduction




(a) Input edges (b) Box tree with relevant info (c) Created pattern

Figure 5.5: Demonstration of how *process_box_tree*() works. (a) shows the initial two edges. (b) shows the box tree that is fetched from cache along with the information about ports as tuples (the first two items are indices for ports of the initial two edges, the third item is the negation flag, the fourth item is the recursion flag). (c) shows the ABDD pattern that is produced by the procedure, nodes in grey are created by the algorithm. The returned edge is the short one on top. Note the two Apply calls used to build the node $g$ on the bottom right.

order (box order $\Gamma$), whether or not some target nodes are terminals or not, and in case of boxes with multiple ports, what are the variables of the nodes.

# Chapter 6

# Experimental Results

The following chapter overviews the implementation in brief and mainly discusses the achieved results. Some of the results were originally presented in [24] (BLIF and DIMACS node counts). Additional benchmarks were analyzed to highlight some potential drawbacks of ABDDs. On top of this, an experiment was performed to highlight how representations of certain Boolean functions using different BDD models grow in size after iteratively using Apply.

### Implementation notes

Building on the work from [24], which was implemented in Python [23] as a proof of concept of canonization algorithms (and contains some important functionality for working with tree automata), the implementation of ABDD Apply was also done in Python (3.12.3), on top of the source code from the initial work. For debugging and presentation purposes, the `graphviz` library [18] was utilized to convert instances of tree automata, UBDAs, and ABDDs into images in DOT format [14]. Python allows for quick prototyping, which was especially useful during implementation of materialization, which went through multiple stages before a working version was obtained. However, it has a major drawback, which is its memory footprint and performance (especially noticeable during canonization algorithms, like normalization and folding). For these reasons, not many larger benchmarks could be evaluated (larger than a few tens of thousands), which was also the case in the initial thesis. The possible improvements will be discussed later.

The Apply algorithm is implemented in the `apply/` module, which contains three submodules:

- `box_algebra/` – pregenerating all box $\odot$-products,
- `materialization/` – pregenerating all pairs of predicate sets and materialized patterns for each used box, and
- `pregenerated/` – source files generated by the above-mentioned modules containing `box_tree_cache` and `materialization_recipes` (which explain how to create ABDD sub-patterns, given a box and a set of predicates, i.e. "ingredients").

The representations in different BDD models were obtained using the same canonization algorithms as ABDDs, just with different box orders: BDDs ($\mathtt{X}$), ZBDDs ($\mathtt{H_0}$), TBDDs ($\mathtt{X}, \mathtt{H_0}$), CBDDs ($\mathtt{X}, \mathtt{H}_\oplus$), CZDDs ($\mathtt{H_0}, \mathtt{X}$), ESRBDDs ($\mathtt{L_0}, \mathtt{H_0}, \mathtt{X}$), ABDDs (see Figure 4.1).

Even though the reduction rules described by boxes are the same as the ones used in any of the mentioned models, the node counts obtained after the canonization may slightly differ from the node counts obtained using the original implementations of the models.

- Tagged edges in TBDDs [34] can combine ABDDs' two reduced edges (between 3 nodes) into one edge (between two nodes) which uses a "tag".
- CBDDs and CZDDs [8] store the information about reduction chains within the nodes themselves, not the edges (as in ABDDs), so the reduction information about the second rule would be inside the node.
- ESRBDDs' [4] reduction algorithm does not describe the order in which the 3 reduction rules are applied, so that might also not reflect the exact node counts as in the original implementation. The reduction algorithm (in other models, too) probably explores the graph's structure bottom-up, while the folding algorithm used during ABDD canonization works top-down, which is another difference.

### Benchmarks description

Three types of benchmarks were used to compare the models:

- hardware circuits from LGSynth91 [38] in Berkeley Logic Interchange Format [5] (BLIF), downloaded from [16] – input BDDs were built using the `BuDDy` library [33],
- logic formulae in the DIMACS format from SATLIB (1000 CNF instances, satisfiable, 20 variables, 91 clauses), downloaded from [30], and
- $N$-queens problem instances, where $N \in \{4, 5, 6, 7, 8\}$, built using [32].

These benchmarks were chosen to facilitate best cases of use for different models. In the hardware circuits, the `X` rule (BDDs) is more advantageous than $\mathtt{H_0}$ (ZBDDs). In $N$-queens problems, the opposite is true. $N$-queens is the ideal scenario for choosing ZBDDs, as it fits all the criteria for efficient representation in that model (see Section 2.6).

## 6.1 Node count comparison

On the analyzed benchmarks, in general, ABDDs performed the best (especially the hardware circuit benchmarks). In some cases of DIMACS benchmarks, CZDDs and ZBDDs worked better, where using $\mathtt{H_0}$ reduction before $\mathtt{L_0}$ was more efficient. The same phenomenon can be seen also in $N$-queens instances. All models that start their folding with the $\mathtt{H_0}$ rule (ZBDD, TBDD, CZDD) outperform every other model. Since the reductions in ABDDs started with the $\mathtt{L_0}$ rule, which does not naturally work well with the character of the benchmarks, the reduction effect was weaker, but still significantly stronger than using just the "don't care" rule of the classic BDDs. In ABDDs, during folding, $\mathtt{L_0}$ reductions were applied to many places where using $\mathtt{H_0}$ would be more efficient, so when it came to trying out $\mathtt{H_0}$ reductions, most attempts failed, which led to the higher node count. This is the drawback of the way folding is implemented (trying one rule everywhere before trying another rule).

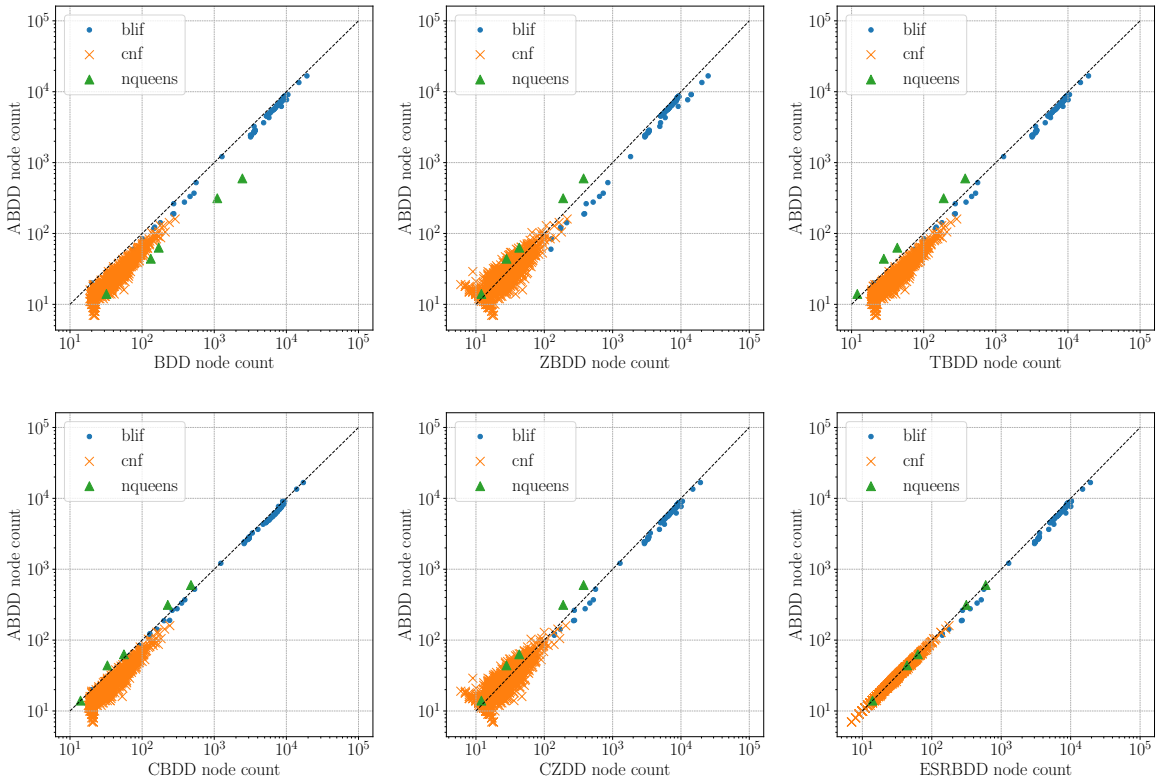



Figure 6.1: Node count comparison of BLIF, DIMACS, and $N$-queens benchmarks.

## 6.2 Examining the growth of progressive Apply usage

Since ABDDs are currently implemented as a Python prototype (proof of concept), and most of the libraries of other models utilize highly optimized, multi-threaded C/C++ code, comparing the runtime of ABDD Apply implementation to those would not lead to interesting results. However, what can be compared is how different models grow in node counts, when multiple Apply operations are used to build a function (such as when using conjunction of clauses during the building of a CNF DIMACS benchmark). The experiment was designed in the following way. After each clause of some DIMACS CNF benchmark was joined using the ABDD's Apply implementation, the size of the canonical form of every model was measured. The clauses were processed and merged with the result until the canonization timed out. If the time out happened after 20 clauses were processed, then only the records of the first 20 Boolean functions' sizes of a partially processed DIMACS benchmark were recorded.

The following experiment showed that ABDDs kept the node counts the lowest during the process. Usually with these benchmarks, after all clauses are processed, the formula is satisfiable only for a few assignments, i.e. the graph has only a few paths to terminal nodes and its representation is small. However, before the final representation of a formula is built, the graph first quickly grows in size, and after around 20-40 clauses are analyzed, the size starts to decrease as many clauses cancel out and simplify the representation. During the graph's growth, especially the "middle part" of the graph allowed more and more places for $\mathtt{L}_{\oplus}, \mathtt{H}_{\oplus}$ reductions, which are not included in other models (except CBDDs in a sense). This explains why ABDDs had the lowest aggregated node counts during the whole process,

followed by CBDDs. ZBDDs performed the worst in this case. First 100 of the DIMACS benchmarks (the same ones as in the previous experiment) were used in this experiment.

The graphs in Figure 6.2 show the aggregated node counts (across all iterations of producing one benchmark, until a 50 second timeout during canonization was reached) of all models compared to ABDDs. The Figures 6.3 and 6.4 then show how the node count develops across iterations of one fully-built DIMACS benchmark. Overall, ABDDs performed best, CBDDs were 2nd most compact (thanks to the utilization of OR-chains, which are simulated using the $\mathtt{H}_{\oplus}$ box). Finally, Table 6.1 contains the aggregated results of all experiments (BLIF, DIMACS, $N$-queens benchmarks, and aggregated node counts comparing progressive Apply usage)

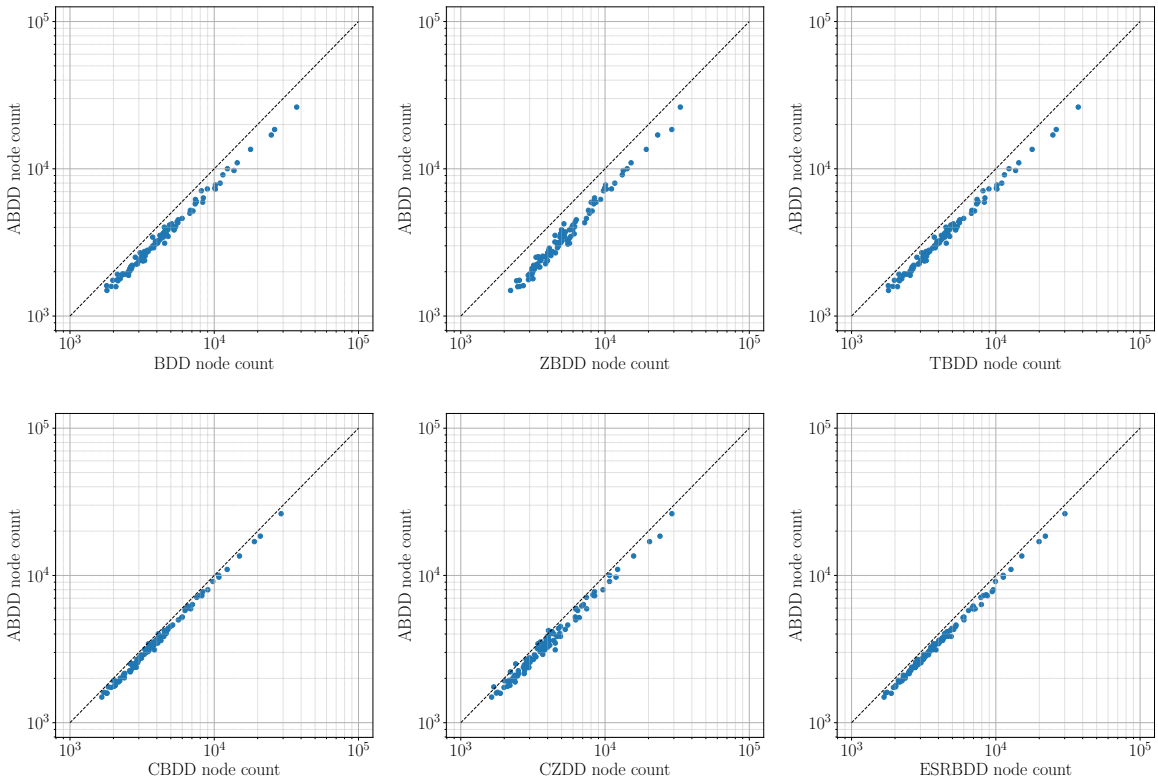



Figure 6.2: Node counts of the 100 tested CNF DIMACS benchmarks aggregated as more clauses were processed. Each dot represents one benchmark, but sums node counts during processing of clauses (essentially each iteration can represent a different Boolean function, timeout usually happened after 15-20 iterations).

Table 6.1: Summary of aggregated node counts for each benchmark type. The row labeled $c$-DIMACS shows the aggregated node counts of the cumulative Apply usage during the build of the 100 DIMACS benchmarks. The last row shows the percentage (averaged over each benchmark) of nodes that were reduced by using ABDD as opposed to the particular model during the cumulative Apply usage.

| | BDD | ZBDD | TBDD | CZDD | CBDD | ESRBDD | **ABDD** |
|---|---|---|---|---|---|---|---|
| C432 | 2009 | 2821 | 2007 | 2023 | 1578 | 1958 | **1438** |
| C880 | 70645 | 94020 | 70645 | 70660 | 62608 | 70327 | **61414** |
| C1355 | 263520 | 255456 | 263520 | 255392 | 258203 | 255360 | **231640** |
| C1908 | 75111 | 72383 | 75095 | 70469 | 61918 | 72587 | **56598** |
| DIMACS | 45517 | 28591 | 45041 | 27845 | 42002 | 27676 | **26713** |
| $N$-queens | 3890 | **648** | **648** | **648** | 806 | 1033 | 1033 |
| $c$-DIMACS | 560790 | 633361 | 560709 | 497927 | 479900 | 496740 | **432260** |
| Improvement | 20.86 % | 33.08 % | 20.85 % | 12.51 % | 10.11 % | 12.41 % | - |

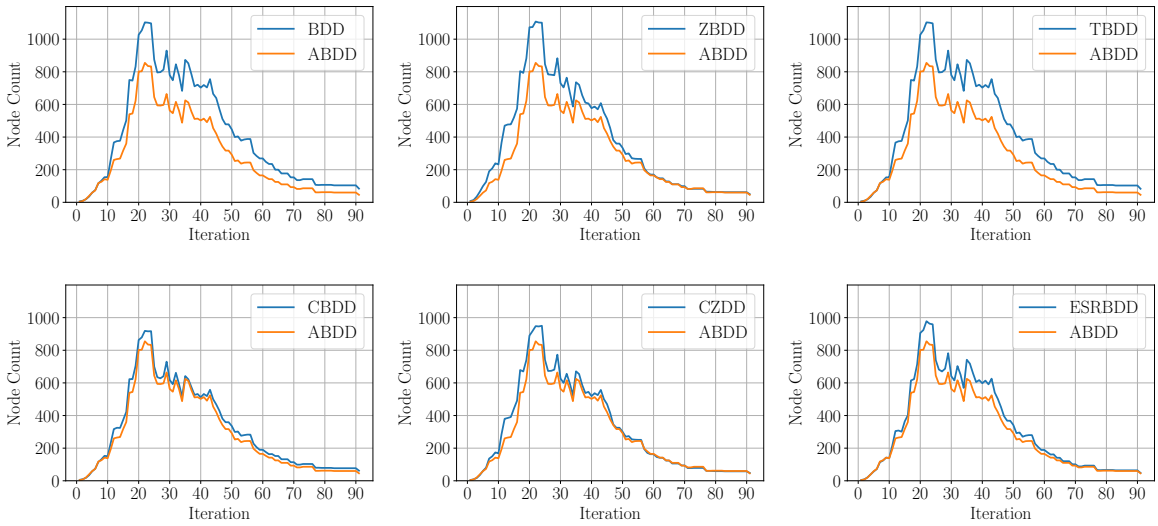



Figure 6.3: An overview of the graph size development of one DIMACS benchmark (`uf20-011`) after clauses were progressively merged with the result.

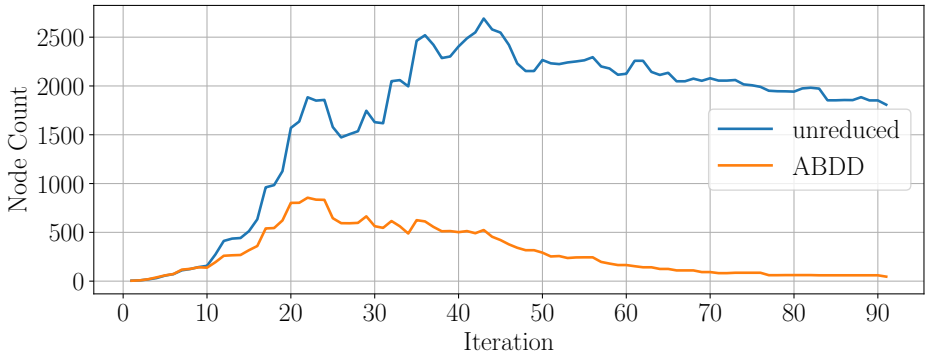



Figure 6.4: Graph size development during building the DIMACS benchmark (`uf20-011`) which aims to compare the reduced (ABDD) and unreduced (before canonization) results of Apply.

# Chapter 7

# Conclusion

This thesis expands on the initial work on automata-based binary decision diagrams [24] where the data structure was first introduced, along with canonization procedures needed to obtain a reduced (or canonical) form. One major drawback of the original work was that the process of combining two (or more) ABDDs using Boolean operators (function Apply) required unfolding the structure and unwinding the loops, making the worst case complexity exponential (with regards to the number of variables).

The work presented here makes the complexity of the Apply algorithm only depend on the graph sizes of the two input Boolean functions represented as ABDDs. The thesis explains how the algorithm works, the techniques used that make it possible to directly combine the reduction rules without needing to expand the structure of the inputs, such that an increase in the node count proportional to the number of variables skipped is avoided.

The current approach keeps the reduction rates consistent with the results from [24]. However, one of the major drawbacks is that the proposed implementation of Apply lacks mechanisms for on-the-fly canonization (i.e. reduction of the newly introduced nodes). The obtained result has to undergo the canonization process (unfolding, normalization, folding), which comes with its own drawbacks.

The automata-based framework is a novel approach and shows promise, yielding around 10-20 % better reduction rates than the most popular decision diagram models (BDDs, ZBDDs). Since the complexity of Apply is proportional to the sizes of the input graphs, in theory, Apply on ABDDs should potentially outperform these models. The main feature of ABDDs is the fact that they are flexible, i.e. allow for introduction of new rules, and changing the order of existing rules, which can produce even better results for specific cases. These factors make ABDDs an interesting area of research.

## 7.1 Future work

The main drawback of ABDD Apply is the need for post-hoc canonization. Finding a way to perform the canonization on-the-fly, such that when a node is created, along with its low and high edges, some systematic check could be performed. This check would try to search for an equivalent, canonical representation for the newly created node, and if the search is successful, then the canonical representation is returned (which should avoid unnecessary growth of the result, as shown in Figure 6.4). For efficiency, all these reducible patterns

with their canonical counterparts should also be precomputed. Right now, it is unclear how such a systematic way of enumerating these reducible patterns would work.

While ABDDs have potential to efficiently reduce the size of binary decision diagrams, there is a lot of room for improvements in terms of performance. Implementing the framework in a language better suited for high performance, such as C/C++, along with utilizing low-level optimization techniques, improving memory performance, and using better data structures, would be a big step forward in making ABDDs more usable. Better performance would also allow for testing on larger and more diverse set of benchmarks.

One of the key drawbacks of the automata-based approach is the complexity of canonization algorithms, especially high memory and performance requirements of normalization (a bottom-up determinization with regards to variables). Furthermore, utilizing different order of reduction rules during folding for different types of benchmarks could also lead to better results.

Since ABDDs provide a framework in which it is possible to introduce new reduction rules, analyzing benchmarks and finding new reducible patterns and specialized rules for them is another possible upgrade.

One of the latest models, CESRBDDs (BDDs with complemented edges and edge-specified reductions [3]), unify nodes with the same meaning (i.e. that represent the same Boolean function $f$) along with nodes with the exact opposite meaning (representing their negation $\neg f$). Edges in such models then utilize complement bits, which determine the interpretation of the edge's target node – either its direct or negated semantics. Currently, ABDDs do not support complemented edges. Implementing complemented edges and merging nodes with opposite semantics is thereby yet another possible way of improving ABDDs which would require revisiting canonization algorithms (especially normalization) to somehow keep information about negation flags for each state from the state sets during the determinization process.

Additionally, there are many other useful operations that manipulate BDDs in practical applications, such as in symbolic model checking, etc. There are also algorithms that reorder variables (such as sifting [29]), directly modifying the BDD structure while retaining canonicity. Providing algorithms for these operations on ABDDs is also a potential area of focus, as is extending ABDDs to include multiple terminal nodes (like in MTBDDs).

# Bibliography


[1] Abdulla, P. A.; Chen, Y.-F.; Holík, L.; Mayr, R. and Vojnar, T. When Simulation Meets Antichains. In: Esparza, J. and Majumdar, R., ed. *Tools and Algorithms for the Construction and Analysis of Systems – 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Springer, 2010, p. 158–174. ISBN 978-3-642-12001-5. Available at: https://doi.org/10.1007/978-3-642-12002-2_14.

[2] Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, 1978, C-27, no. 6, p. 509–516. DOI: 10.1109/TC.1978.1675141. Available at: https://ieeexplore.ieee.org/document/1675141.

[3] Babar, J.; Ciardo, G. and Miner, A. CESRBDDs: binary decision diagrams with complemented edges and edge-specified reductions. *International Journal on Software Tools for Technology Transfer*. Springer Science and Business Media LLC, february 2022, vol. 24, no. 1, p. 89–109. DOI: 10.1007/s10009-021-00640-0. ISSN 1433-2787. Available at: http://dx.doi.org/10.1007/s10009-021-00640-0.

[4] Babar, J.; Jiang, C.; Ciardo, G. and Miner, A. Binary Decision Diagrams with Edge-Specified Reductions. In: Vojnar, T. and Zhang, L., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, p. 303–318. ISBN 978-3-030-17465-1.

[5] Berkeley Logic Interchange Format (BLIF). 1992. University of California, Berkeley. Available at: https://course.ece.cmu.edu/~ee760/760docs/blif.pdf.

[6] Bollig, B. and Wegener, I. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 1996, vol. 45, no. 9, p. 993–1002. DOI: 10.1109/12.537122. Available at: https://ieeexplore.ieee.org/document/537122.

[7] *Boolean Function*. Encyclopedia of Mathematics. Available at: http://encyclopediaofmath.org/index.php?title=Boolean_function&oldid=44593. Adapted from *Encyclopedia of Mathematics* – ISBN 1402006098.

[8] Bryant, R. E. Chain Reduction for Binary and Zero-Suppressed Decision Diagrams. In: Beyer, D. and Huisman, M., ed. *Tools and Algorithms for the Construction and Analysis of Systems – 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*. Springer, 2018, vol.

10805, p. 81–98. Lecture Notes in Computer Science. ISBN 978-3-319-89959-6. Available at: https://doi.org/10.1007/978-3-319-89960-2_5.

[9] Bryant, R. E. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 1986, C-35, no. 8, p. 677–691. DOI: 10.1109/TC.1986.1676819. Available at: https://ieeexplore.ieee.org/document/1676819.

[10] Bryant, R. E. Binary Decision Diagrams: An Algorithmic Basis for Symbolic Model Checking. In: Clarke, E. M.; Henzinger, T. A.; Veith, H. and Bloem, R., ed. *Handbook of model checking*. 1st ed. Basel, Switzerland: Springer International Publishing, June 2018, p. 191–218. ISBN 978-3-319-10575-8. Available at: https://www.cs.cmu.edu/~bryant/pubdir/hmc-bdd18.pdf.

[11] Chaki, S.; Gurfinkel, A. and Strichman, O. Decision diagrams for linear arithmetic. In: *2009 Formal Methods in Computer-Aided Design*. 2009, p. 53–60. Available at: https://ieeexplore.ieee.org/document/5351143.

[12] Comon, H.; Dauchet, M.; Gilleron, R.; Löding, C.; Jacquemard, F. et al. *Tree Automata Techniques and Applications*. 2007. Available at: http://www.grappa.univ-lille3.fr/tata.

[13] Coudert, O. Solving graph optimization problems with ZBDDs. In: *Proceedings European Design and Test Conference. ED & TC 97*. 1997, p. 224–228. Available at: https://ieeexplore.ieee.org/document/582363.

[14] Graphviz Project. *DOT language documentation*. 2025. Available at: https://graphviz.org/doc/info/lang.html.

[15] Een, N.; Mishchenko, A. and Sörensson, N. Applying Logic Synthesis for Speeding Up SAT. In: Marques Silva, J. and Sakallah, K. A., ed. *Theory and Applications of Satisfiability Testing – SAT 2007*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, p. 272–286. ISBN 978-3-540-72788-0. Available at: https://people.eecs.berkeley.edu/~alanmi/publications/2007/sat07_map.pdf.

[16] Fišer, P. and Schmidt, J. A Comprehensive Set of Logic Synthesis and Optimization Examples. In: *Proceedings of the 12th International Workshop on Boolean Problems (IWSBP)*. Freiberg, Germany: Freiberg University of Mining and Technology, Institute of Computer Science, September 2016, p. 151–158. ISBN 9783860125403. Available at: https://ddd.fit.cvut.cz/www/prj/Benchmarks/.

[17] Friedman, S. and Supowit, K. Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computers*, 1990, vol. 39, no. 5, p. 710–713. DOI: 10.1109/12.53586. Available at: https://ieeexplore.ieee.org/document/53586.

[18] *Graphviz: Python interface for Graphviz DOT language*. 2025. Available at: https://pypi.org/project/graphviz/.

[19] James, G.; Witten, D.; Hastie, T. and Tibshirani, R. *An introduction to statistical learning*. 2nd ed. New York, NY: Springer, 2013. Available at: https://www.karlin.mff.cuni.cz/~pesta/NMFM334/StatLearning/Book2nd/ISLRv2_website.pdf.

[20] Larsen, K. G.; Pearson, J.; Weise, C. and Yi, W. Clock difference diagrams. *Nordic J. of Computing*. FIN: Publishing Association Nordic Journal of Computing, september 1999, vol. 6, no. 3, p. 271–298. DOI: 10.5555/774455.774459. ISSN 1236-6064. Available at: https://vbn.aau.dk/files/425046823/CDD_26pages_nordic_journal_of_computing_1999.pdf.

[21] Lee, C. Y. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 1959, vol. 38, no. 4, p. 985–999. DOI: 10.1002/j.1538-7305.1959.tb01585.x. Available at: https://ieeexplore.ieee.org/document/6768525.

[22] Liaw, H.-T. and Lin, C.-S. On the OBDD-representation of general Boolean functions. *IEEE Transactions on Computers*, 1992, vol. 41, no. 6, p. 661–664. DOI: 10.1109/12.144618. Available at: https://ieeexplore.ieee.org/document/144618.

[23] Maťufka, J. *Automata-based binary decision diagram implementation GitHub repository*. 2025. Available at: https://github.com/Jany26/tree-aut-lib.

[24] Maťufka, J. *New Techniques for Compact Representation of Boolean Functions*. Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondřej Lengál, Ph.D. Available at https://www.vut.cz/en/students/final-thesis/detail/146866.

[25] Minato, S.-i. Zero-suppressed BDDs for set manipulation in combinatorial problems. In: *Proceedings of the 30th International Design Automation Conference*. New York, NY, USA: Association for Computing Machinery, 1993, p. 272–277. DAC '93. ISBN 0897915771. Available at: https://doi.org/10.1145/157485.164890.

[26] Minato, S.-i. Zero-suppressed BDDs and their applications. *International Journal on Software Tools for Technology Transfer*. Springer Science and Business Media LLC, may 2001, vol. 3, no. 2, p. 156–170. DOI: 10.1007/s100090100038. ISSN 1433-2779. Available at: http://dx.doi.org/10.1007/s100090100038.

[27] Minato, S.-i.; Uno, T. and Arimura, H. LCM over ZBDDs: Fast Generation of Very Large-Scale Frequent Itemsets Using a Compact Graph-Based Representation. In: *Advances in Knowledge Discovery and Data Mining*. Springer Berlin Heidelberg, 2008, p. 234–246. ISBN 9783540681250. Available at: http://dx.doi.org/10.1007/978-3-540-68125-0_22.

[28] Møller, J.; Lichtenberg, J.; Andersen, H. R. and Hulgaard, H. Difference Decision Diagrams. In: *Computer Science Logic*. Springer Berlin Heidelberg, 1999, p. 111–125. ISBN 9783540481683. Available at: http://dx.doi.org/10.1007/3-540-48168-0_9.

[29] Rudell, R. Dynamic variable ordering for ordered binary decision diagrams. In: *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. 1993, p. 42–47. Available at: https://ieeexplore.ieee.org/document/580029.

[30] Hoos, H. *SATLIB Benchmark Suite*. 2011. Available at: https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html.

[31] Shannon, C. E. A symbolic analysis of relay and switching circuits. *Transactions of the American Institute of Electrical Engineers*. Institute of Electrical and Electronics Engineers (IEEE), december 1938, vol. 57, no. 12, p. 713–723. DOI: 10.1109/t-aiee.1938.5057767. ISSN 0096-3860. Available at: http://dx.doi.org/10.1109/T-AIEE.1938.5057767.

[32] Sølvsten, S. *Benchmarking Suite for BDD packages*. 2024. Available at: https://github.com/SSoelvsten/bdd-benchmark.

[33] Sølvsten, S. *BuDDy: Binary Decision Diagram Library*. 2025. Available at: https://github.com/SSoelvsten/buddy. Includes links to original documentation and PhD thesis. Doxygen documentation available at: https://buddy.sourceforge.net/manual/main.html.

[34] van Dijk, T.; Wille, R. and Meolic, R. Tagged BDDs: Combining reduction rules from different decision diagram types. In: Stewart, D. and Weissenbacher, G., ed. *2017 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2017, p. 108–115. ISBN 978-3540001164. Available at: https://ieeexplore.ieee.org/document/8102248.

[35] van Dijk, T. et al. *Sylvan: Multi-core Decision Diagram (BDD/LDD) implementation*. 2023. Available at: https://github.com/utwente-fmt/sylvan.

[36] van Dijk, T.; Laarman, A. and van de Pol, J. Multi-Core BDD Operations for Symbolic Reachability. *Electronic Notes in Theoretical Computer Science*, 2013, vol. 296, p. 127–143. DOI: https://doi.org/10.1016/j.entcs.2013.07.009. ISSN 1571-0661. Available at: https://www.sciencedirect.com/science/article/pii/S1571066113000406. Proceedings the Sixth International Workshop on the Practical Application of Stochastic Modelling (PASM) and the Eleventh International Workshop on Parallel and Distributed Methods in Verification (PDMC).

[37] Wegener, I. The size of reduced OBDD's and optimal read-once branching programs for almost all Boolean functions. *IEEE Transactions on Computers*, 1994, vol. 43, no. 11, p. 1262–1269. DOI: 10.1109/12.324559. Available at: https://ieeexplore.ieee.org/abstract/document/324559.

[38] Yang, S. *Logic Synthesis and Optimization Benchmarks User Guide: Version 3.0*. Microelectronics Center of North Carolina (MCNC), 1991. Available at: https://ddd.fit.cvut.cz/www/prj/Benchmarks/LGSynth91.pdf.