INSTITUTE FOR INFORMATICS

LMU MUNICH

Master's Thesis in Computer Science

Supporting a CDCL Sat Solver by BDD Methods

Lydia Kondylidou

INSTITUTE FOR INFORMATICS

LMU MUNICH

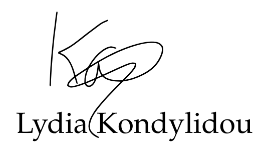
Master's Thesis in Computer Science

Supporting a CDCL Sat Solver by BDD Methods

Unterstützung eines CDCL Sat Solvers durch BDD Methoden

Author:	Lydia Kondylidou
Supervisor:	Dr. Jan Johannsen
Submission Date:	31.01.2023

I confirm that this master's thesis in computer science is my own work and I have documented all sources and material used.



Munich, 31.01.2023 Lydia Kondylidou

Acknowledgments

I would like to thank the following people, without whom I would not have been able to complete this research and without whom I would not have made it through my Master's degree!
My supervisor Dr. Jan Johannsen, whose insight and knowledge of the subject matter guided me through this project. And special thanks to Florian Leimgruber, whose contribution has been beneficial, as my work is a continuation of his Bachelor thesis.
My most enormous thanks to my family for all the unconditional support throughout these intense academic years. Thank you for the sacrifices you made for me to be able to follow my dreams, for encouraging me never to give up, even though there were challenging moments of endless hard work and most importantly, thank you for understanding me and pressuring me to believe in me.

Abstract

Boolean satisfiability (SAT) is one of the most important problems of theoretical computer science with many practical applications in which Boolean Satisfiability (SAT) solvers are used in the background as high-performance reasoning engines. These applications include automated planning and scheduling, formal verification, and automated theorem proving. In the last decades, the performance of state-of-the-art SAT solvers has increased dramatically, thanks to the invention of advanced heuristics, preprocessing and inprocessing techniques, and data structures that allow efficient implementation of search space pruning. The increasing popularity of SAT in verification and synthesis encourages the search for additional speed-ups. This paper presents a new approach to support a Conflict Driven Clause Learning (CDCL) SAT solver with Binary Decision Diagram (BDD) techniques. While SAT and BDD techniques are often presented as mutually exclusive alternatives, this work points out that both can be improved by running in parallel and exchanging information. In this thesis, a BDD library is implemented in Rust, designed to support the latest version of the Glucose SAT solver. The proposed methods are based on efficient communication between the two architectures. Several benchmarks from the SAT competitions are run both on Glucose alone and Glucose with the contribution of the BDDs, and the results are compared. Eventually, the experiments show that the performance of Glucose is improved when running in parallel with the BDD solver.

Kurzfassung

Die boolesche Erfüllbarkeit (SAT) ist eines der wichtigsten Probleme der theoretischen Informatik mit vielen praktischen Anwendungen, bei denen Boolean Satisfiability (SAT) Solver im Hintergrund als leistungsstarke Argumentationsmaschinen eingesetzt werden. Zu diesen Anwendungen gehören die automatisierte Planung und Terminierung, die formale Verifikation und das automatisierte Theorembeweis. In den letzten Jahrzehnten hat sich die Leistung moderner SAT Solver dank der Erfindung fortschrittlicher Heuristiken, Preprocessing- und Inprocessing-Techniken und Datenstrukturen, die eine effiziente Implementierung von Suchraumbeschneidung ermöglichen, drastisch erhöht. Die zunehmende Beliebtheit von SAT Solvern in der Verifikation und Synthese ermutigt die Suche nach zusätzlichen Beschleunigungstechniken. In dieser Thesis wird ein neuer Ansatz zur Unterstützung eines Conflict Driven Clause Learning (CDCL) SAT Solvers mit Binary Decision Diagram (BDD)-Techniken präsentiert. Während SAT- und BDD-Techniken oft als sich gegenseitig ausschließende Alternativen dargestellt werden, zeigt diese Arbeit, dass beide verbessert werden können, wenn sie parallel laufen und Informationen austauschen. In dieser Arbeit wird eine BDD-Bibliothek in Rust implementiert, die zur Unterstützung der neuesten Version des Glucose SAT Solvers entwickelt wurde. Die vorgeschlagenen Methoden beruhen auf einer effizienten Kommunikation zwischen den beiden Architekturen. Mehrere Benchmarks aus den SAT Wettbewerben werden sowohl auf Glucose allein als auch auf Glucose mit dem Beitrag der BDDs ausgeführt, und die Ergebnisse werden verglichen. Letztendlich zeigen die Experimente, dass die Leistung von Glucose verbessert wird, wenn es parallel mit dem BDD Solver läuft.

Contents

Acknowledgments **iii**

Abstract **iv**

Kurzfassung **v**

1 Introduction **1**
- 1.1 Motivation and Overview . . . 1
  - 1.1.1 Motivation . . . 1
  - 1.1.2 Related Work . . . 2
  - 1.1.3 Overview . . . 2

2 Preliminaries **3**
- 2.1 Boolean Expressions . . . 3
- 2.2 Normal Forms . . . 4
  - 2.2.1 Conjuctive Normal Form . . . 4
  - 2.2.2 Disjunctive Normal Form . . . 4
- 2.3 Binary Decision Diagrams . . . 5
  - 2.3.1 Reduced Ordered Binary Decision Diagrams . . . 8
- 2.4 The programming language Rust . . . 8

3 CDCL Sat Solver **11**
- 3.1 DPLL . . . 11
- 3.2 Unit Propagation . . . 11
- 3.3 Decision Level . . . 12
- 3.4 CDCL . . . 12
  - 3.4.1 Restarts . . . 13
  - 3.4.2 Conflict Analysis . . . 13
  - 3.4.3 Clause Learning . . . 17

4 Parallel BDD Library in Rust to support the CDCL process **18**
- 4.1 Existing BDD Libraries . . . 18
  - 4.1.1 CUDD . . . 18
  - 4.1.2 BuDDy . . . 18
  - 4.1.3 Sylvan . . . 19
- 4.2 BDD Library Design . . . 19

4.3 Modular Components 20
4.3.1 DIMACS Parser 20
4.3.2 Variable Ordering 21
4.3.3 BDD Creation 22

5 Communication between the architectures **32**
5.1 Glucose 32
5.2 Data Parallelism 33
5.3 Information Exchange 34

6 Performance and Measurements **37**
6.1 Benchmarks 37
6.1.1 Test Cases 38
6.2 Testing Environment 38
6.3 Runtime and Measurements 38
6.4 Code and Data 39
6.5 Analysis 39

7 Conclusion **46**

8 Future Work and Suggestions **47**

List of Figures **49**

Glossary **50**

Acronyms **51**

Bibliography **52**

1 Introduction

1.1 Motivation and Overview

1.1.1 Motivation

Solvers for Boolean Satisfiability (SAT) have become increasingly influential in recent decades. The performance of state-of-the-art SAT solvers has increased dramatically, thanks to the invention of advanced heuristics, preprocessing, and inprocessing techniques and data structures that allow efficient implementation of search space pruning. The above was proven in the last SAT competitions, where the focus lies on identifying new challenging benchmarks, promoting new solvers for the propositional satisfiability problem (SAT), and comparing them with state-of-the-art solvers. Modern SAT solvers have also shown remarkable results in real-world applications. They have significantly impacted fields, including software verification, program analysis, constraint solving, artificial intelligence, electronic design automation, and operations research.
Modern SAT solvers' success stems from their ability to quickly learn new constraints from infeasible search states via Conflict Driven Clause Learning (CDCL). The architecture of today's SAT solvers, combining unit propagation with rapid restarts and CDCL, focuses on techniques with very low overhead and maximizes the number of search nodes that can be processed per second [1]. As this has been beneficial, new ways to support this process are being investigated.
The next natural step in the development of SAT solvers was parallelization. A ubiquitous approach to designing a parallel SAT solver is to run several instances of a sequential SAT solver with different settings (or several different SAT solvers) on the same problem in parallel. If any solvers succeed in finding a solution, all the solvers terminate. The solvers also exchange information mainly in the form of learned clauses [2]. This approach is referred to as portfolio-based parallel SAT solving and was first used in the SAT solver ManySat [3]. Another approach is to run different types of solvers in parallel, for example, a BDD solver and a CDCL solver.
The purpose of this Master thesis is to support the CDCL process by exchanging information, mainly learnt clauses not derived from unit propagation. The idea behind this is that clauses extracted from the construction of BDDs, representing a given propositional formula, can substantially reduce the runtime and enhance the performance of a competitive SAT solver when added to the original formula. When running in parallel with the CDCL SAT solver, the BDD obtains sets of learnt clauses while still being constructed by applying a top-down compilation scheme for those nodes that do not lead to a satisfying solution. These clauses are marked as witness clauses and are sent to the CDCL SAT solver. Since a node in a BDD

can represent multiple partial assignments, a single witness clause generated in this way is as strong as multiple witness clauses derived from these separate partial assignments. The clauses acquired from the SAT solvers' conflict analysis are also sent back to the BDD and added to the original propositional formula to boost the BDDs performance likewise.
BDDs that exactly represent a given Conjunctive Normal Form (CNF) formula are well known to grow exponentially in general. The exponential growth of the solvers has significantly limited the success of BDD- based techniques for SAT solving. The BDD is being approximated during the construction process to overcome this limitation. BDD approximation is the process of deriving from a given BDD another BDD more minor, and whose function is at a low Hamming distance from the input BDD.
Finally, the results of computational experiments performed to evaluate the use of BDDs to support a CDCL SAT Solver are presented and analyzed. The results show that when the BDD and the CDCL SAT solver run in parallel, the overall process becomes more efficient. In particular, some problems, not terminating when the CDCL SAT solver was running alone, finished before the given timeout with the support of the BDDs. However, running both the CDCL SAT solver and the BDD solver in parallel means doing more work, and for that, the improvement in the overall procedure shown in the results could be better, which leaves room for improvement. Nonetheless, the qualitative contribution of the BDD demonstrates excellent potential for inclusion in SAT solvers, and several suggestions for doing so, as well as suggestions for specific improvements, are proposed in chapter 8.

1.1.2 Related Work

It is important to note that this Master's thesis is a continuation of the bachelor thesis of Florian Leimgruber [4]. In his thesis, Florian implemented his own BDD library and SAT solver to revisit BDD-based approaches and investigate how valuable BDDs are in SAT solving. Significantly, the performance of parallel SAT solvers that use BDDs to support a CDCL solver was investigated, and BDD approximation algorithms were used to limit the BDD sizes. On the one hand, BDDs were used to derive lexicographic search space constraints, while on the other hand, an algorithm was developed to derive small clauses from BDDs. The second approach has advantages in selected combinatorial problems.

1.1.3 Overview

This thesis is structured as follows. Chapter 2 briefly introduces Boolean expressions, normal forms, and Binary Decision Diagram (BDD)s. Chapter 3 is dedicated to CDCL SAT Solvers. In chapter 4, the BDD library implemented in this thesis is presented, as well as its key ideas and algorithms. Chapter 5 focuses on the communication between the two architectures, namely the CDCL SAT Solver, Glucose, and the BDD library. The code and data, performance and measurements, and an analysis of the test results are presented in Chapters 6 and 7. In Chapter 8, future work and suggestions are considered.

2 Preliminaries

This section introduces Binary Decision Diagram (BDD)s and explains their properties. Especially, it provides some background knowledge and gives examples of their construction. More details can be ecnountered in Bryant's original paper on Reduced Ordered Binary Decision Diagram (ROBDD) [5]. The implementation of BDDs in this Master thesis and the core ideas and algorithms will be presented in the following chapters. In the end, this section gives an overview of the programming language Rust used in this Master's project.

2.1 Boolean Expressions

The classical calculus for dealing with truth values consists of Boolean variables x, y, ..., the constants *true* 1 and *false* 0, the operators of *conjunction* $\wedge$, *disjunction* $\vee$, *negation* $\neg$, *implication* $\implies$, and *bi* $-$ *implication* $\iff$ which together form the Boolean expressions. Sometimes the variables are called *propositional variables*, and the Boolean expressions are then known as *Propositional Logic*. Formally, Boolean expressions are generated from the following grammar:

$$(p,q) ::= x|0|1|\neg p|p \wedge q|p \vee q|p \implies q|p \iff q,$$

where x ranges over a set of Boolean variables. This is called the *abstract syntax* of Boolean expressions. A Boolean expression with variables $x_1, ..., x_n$ denotes for each assignment of truth values to the variables itself a truth value according to the standard truth tables as seen in figure 2.1.

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

p	q	$p \implies q$
T	T	T
T	F	F
F	T	T
F	F	T

p	q	$p \vee q$
T	T	T
T	F	F
F	T	F
F	F	T

Figure 2.1: Truth Tables

Truth assignments are written as sequences of assignments of values to variables, e.g., $[0/x_1, 1/x_2, 0/x_3, 1/x_4]$, which assigns 0 to x_1, 1 to x_2, 0 to x_3, 1 to x_4. With this particular truth assignment, the above expression has value 1, whereas $[0/x_1, 1/x_2, 0/x_3, 0/x_4]$ yields 0. The set of truth values is often denoted $B = 0, 1$. If an ordering of the variables of Boolean expressions f is fixed, f can be viewed as defining a function from B^n to B where n is the

number of variables. The particular ordering chosen for the variables is essential for defining the function. Let us consider, for example, the expression $x \implies y$. If the ordering $x < y$ is chosen, then this is the function $f(x, y) = x \implies y$, true if the first argument implies the second, but if the ordering $y < x$ is chosen, then it is the function $f(y, x) = x \implies y$, true if the second argument implies the first. Later, when compact representations of Boolean expressions are considered, such variable orderings play a crucial role, as well as for the BDDs themselves.
Two Boolean expressions f and f' are said to be equal if they yield the same truth value for all truth assignments. A Boolean expression is a *tautology* if it yields true for all truth assignments; it is $satisfiable$ if it yields true for at least one truth assignment [6].

2.2 Normal Forms

2.2.1 Conjuctive Normal Form

Conjunctive Normal Form (CNF) is an approach to Boolean logic that expresses formulas as conjunctions of clauses or terms with an AND or OR. Each clause connected by a conjunction, or AND, must be either a literal or contain a disjunction or OR operator. For example:

$$(x_1 \vee x_2) \wedge (x_2 \vee x_3) \text{ or } (\neg x_1 \vee x_2) \wedge (x_2 \vee \neg x_3).$$

Literals are seen in CNF as conjunctions of literal clauses and conjunctions that happen to have a single clause. It is possible to convert statements into CNF that are written in another form, such as disjunctive normal form.

2.2.2 Disjunctive Normal Form

Disjunctive Normal Form (DNF) is the normalization of a logical formula in Boolean mathematics. In other words, a logical formula is said to be in disjunctive normal form if it is a disjunction of conjunctions with every variable. Its negation is present once in each conjunction. All disjunctive normal forms are non-unique, as all disjunctive normal forms for the same proposition are mutually equivalent. For example:

$$(x_1 \wedge x_2) \vee (x_2 \wedge x_3) \text{ or } (\neg x_1 \wedge \neg x_2) \vee (x_2 \wedge x_3).$$

Proposition 2.2.0.1. Any Boolean expression is equal to an expression in CNF and an expression in DNF.

Theorem 2.2.1 (Cook–Levin Theorem)**.** The Boolean satisfiability problem is NP-complete. That is, it is in NP, and any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the Boolean satisfiability problem.

2.3 Binary Decision Diagrams

A BDD representation of a Boolean function $f : B^n \to B$, where $B = 0,1$, over a set $X^n = \{x_1, ..., x_n\}$ of Boolean variables is a directed acyclic graph with a vertex (node) set V. Vertices are divided into two types: terminal and non-terminal (internal). A terminal vertex is labeled by zero or one. Each non-terminal vertex v is labeled by a variable $x \in X^n$ and has two children, $low(v), high(v) \in V$, corresponding to whether the variable evaluates to zero or one. The function value is evaluated for a given assignment to the variables by tracing a path from the root to a terminal. The evaluation starts at the root for a given input $m = (m_1, .., m_n)$. At a non-terminal node v with label x_i, if $m_i = 0$, then the outgoing edge corresponding to $low(v)$ is chosen. Otherwise, the edge corresponding to $high(v)$ is chosen [7].
In this section, it is explained how a Binary Decision Diagram (BDD) is derived from a Boolean expression.

The Shannon expansion or decomposition theorem, also known as Boole's expansion theorem, is an identity that allows the expansion of any logic function to be broken down into parts. Formally, is the identity: $f = x \vee f_x \wedge x' \vee f_{x'}$, where f is any Boolean function, x is a variable, x' is the complement of x, and f_x and $f_{x'}$ are f with the argument x set equal to 1 and to 0 respectively. The terms f_x and $f_{x'}$ are sometimes called the positive and negative Shannon cofactors of f with respect to x.

Theorem 2.3.1. Boole's expansion theorem states that for every Boolean function f and for every variable $x_i, i \in \{1..n\}$, where x_i' is the complement of x it holds that: $f(x_1, x_2, \ldots, x_n) = x_1 \vee f(1, x_2, \ldots, x_n) \wedge x_1' \vee f(0, x_2, \ldots, x_n)$.

Example 2.3.1. Let us consider the Boolean expression $f = (x_1 \iff y_1) \wedge (x_2 \iff y_2)$. If Shannon expansions are performed on the expression by selecting in order the variables x_1, y_1, x_2, y_2 and naming the subexpressions as f_i, $i \in \{1..n\}$ we get the following:

$$
\begin{gathered}
f = x_1 \vee f_1 \wedge x_1' \vee f_0 \\
f_0 = y_1 \vee 0 \wedge y_1' \vee f_{00} \\
f_1 = y_1 \vee t_{11} \wedge y_1' \vee 0 \\
f_{00} = x_2 \vee t_{001} \wedge x_1' \vee f_{000} \\
f_{11} = x_2 \vee t_{111} \wedge x_1' \vee f_{110} \\
f_{000} = y_2 \vee 0 \wedge y_1' \vee 1 \\
f_{001} = y_2 \vee 1 \wedge y_1' \vee 0 \\
f_{110} = y_2 \vee 0 \wedge y_1' \vee 1 \\
f_{111} = y_2 \vee 1 \wedge y_1' \vee 0
\end{gathered}
$$

Figure 2.2 shows the expression as a tree. Such a tree is also called a decision tree.

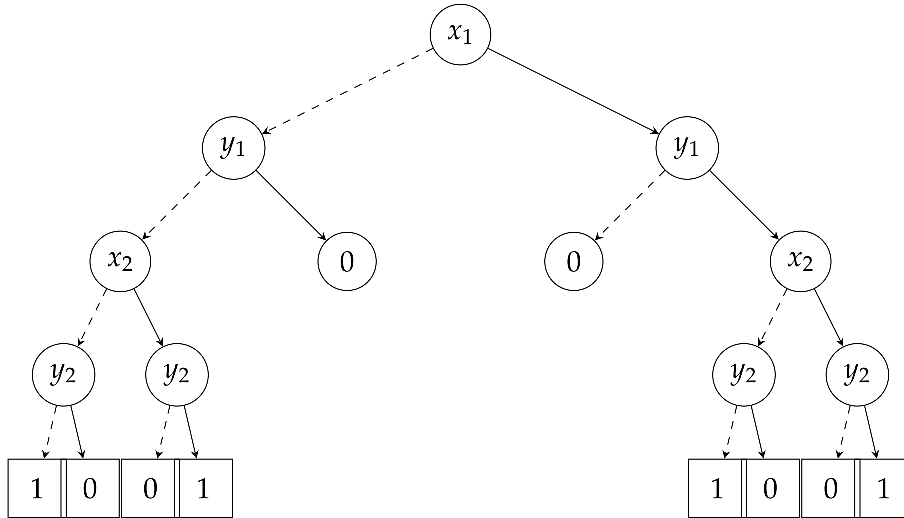


Figure 2.2: A Decision Tree for $(x_1 \iff y_1) \wedge (x_2 \iff y_2)$

Many expressions are identical so that they can be identified and replaced. This results in what is known as a Binary Decision Diagram (BDD), which is no longer a tree of Boolean expressions but a Directed Acyclic Graph (DAG).
Applying the substitutions and reducing the unused expressions, f can be viewed now as follows:

$$
\begin{gathered}
f = x_1 \vee f_1 \wedge x_1' \vee f_0 \\
f_0 = y_1 \vee 0 \wedge y_1' \vee f_{00} \\
f_1 = y_1 \vee t_{00} \wedge y_1' \vee 0 \\
f_{00} = x_2 \vee t_{001} \wedge x_1' \vee f_{000} \\
f_{000} = y_2 \vee 0 \wedge y_1' \vee 1 \\
f_{001} = y_2 \vee 1 \wedge y_1' \vee 0
\end{gathered}
$$

Each subexpression can be viewed as the node of a graph. As explained before, such a node is either *terminal* in the case of the constants 0 and 1 or $non - terminal$. A non-terminal node has a low edge corresponding to the expression bound to the complement of the variable and a high edge corresponding to the expression bound to the variable. Since variables are consistently selected in the same order in the recursive calls during the expansion of the expression, the variables occur in the same orderings on all paths from the root of the BDD. In this situation, the BDD is said to be ordered, an OBDD. Figure 2.3 shows a BDD that is also an OBDD.

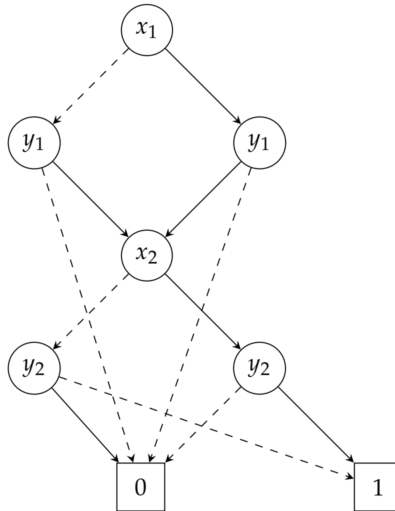


Figure 2.3: An Ordered Binary Decision Diagram with Variable Ordering $x_1 < y_1 < x_2 < y_2$

The ordering of variables chosen when constructing an OBDD impacts the size of the OBDD significantly. Let us now consider the variable ordering $x_1 < x_2 < y_1 < y_2$ instead of $x_1 < y_1 < x_2 < y_2$ as it was before and construct the corresponding OBDD. In figure 2.4, it is shown that the OBDD consists of nine nodes when the variable ordering is $x_1 < x_2 < y_1 < y_2$ and not six nodes as seen in figure 2.3 with the variable ordering $x_1 < y_1 < x_2 < y_2$.

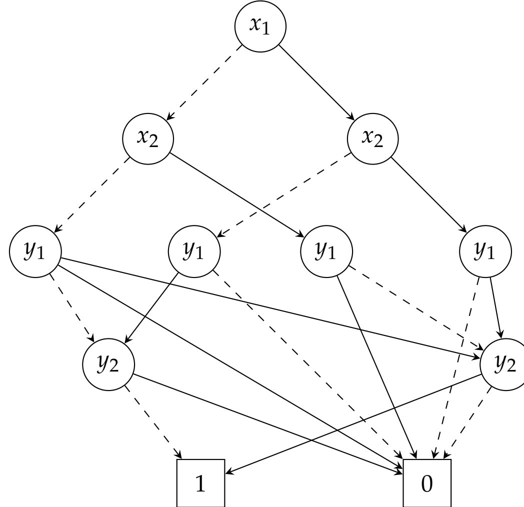


Figure 2.4: An Ordered Binary Decision Diagram with Variable Ordering $x_1 < x_2 < y_1 < y_2$

If no two distinct nodes u and v have the same variable name and low- and high-successor, and if no variable node u has identical low- and high-successor, the OBDD can be called a Reduced Ordered Binary Decision Diagram (ROBDD).

2.3.1 Reduced Ordered Binary Decision Diagrams

The example above showed how a OBDD could be acquired from a Boolean expression by a simple recursive procedure. The characteristics of a ROBDD were also discussed. This Master's project considers ROBDDs as they have some very convenient properties. Specifically, they provide compact representations of Boolean expressions, and there are efficient algorithms for performing all kinds of logical operations on ROBDDs. All the valuable assets of the ROBDDs are centered around the canonicity lemma presented below.

Lemma 2.3.1.1. For any function $f : B^n \to B$ there is exactly one ROBDD u with variable ordering $x_1 < ... < x_n$ such that $f^u = f(x_1, ..., x_n)$, u being a node and f^u the function that maps $(b_1, ..., b_n) \in B^n$ to the truth value of the Boolean expression t, namely $t^u[b_1/x_1, ..., b_n/x_n]$.

An immediate consequence is the following. Since the terminal 1 is a ROBDD for all variable orderings, it is the only ROBDD that is constantly true. So, to check whether an ROBDD is consistently true, it suffices to check whether it is the terminal 1, which is a constant time operation. Similarly, ROBDDs that are constantly false must be identical to the terminal 0. To determine whether two Boolean functions are the same, it suffices to construct their ROBDDs (in the same graph) and check whether the resulting nodes are the same [6]. This problem is NP-complete.

The question arises now how ROBDDs can be constructed. One way is to build an OBDD and reduce it. A graph can be reduced by the repeated application of the following two rules until they are no longer applicable. These rules are:

- Merging Rule: Two isomorphic sub-graphs should be merged.
- Deletion Rule: A vertex (node) whose two branches point to the same vertex should be deleted.

Another way, also researched in this project, is to reduce the OBDD while constructing it. The exact procedure applied in this implementation, beginning from selecting the adequate variable ordering, as well as the core ideas and algorithms to reduce the OBDD during construction, will be described in chapter 4.

2.4 The programming language Rust

Parallel programming, where different parts of a program execute simultaneously, is becoming increasingly important as more computers take advantage of their multiple processors.
In this project, speed and parallelism are of utmost importance. The choice of a programming language dictates the speed of an application program. Rust is one programming language that promises efficient code and is built for parallel programming.
Rust is a programming language, the runtime system of which is open source, that focuses on speed, memory safety, and parallelism. Developers use Rust for a wide range of applications:

Game engines, [1] operating systems, [2] data systems, and browser components [8]. An active community of volunteer coders maintains the Rust programming language and continues to add new enhancements. Mozilla sponsored the Rust open-source project (`https://www.rust-lang.org`) for several years.
Rust makes systems programming accessible by combining speed with efficiency. Using Rust, programmers can make software less prone to bugs and security exploits.
Rust provides thread safety in the form of core language features. In all code that is not marked unsafe, correct access to program data is required and checked by the compiler. The libraries provide more fine-grained synchronization tools. Multithreading is, therefore, a natural choice when working with Rust. A naive implementation starts new threads or processes whenever they are needed. High-performance applications avoid operating system overhead by creating a fixed number of threads or processes in advance. Rust supports coroutines (or asynchronous functions) that can be safely executed in different threads.
One of the most significant benefits of using a systems programming language like Rust is the ability to control low-level aspects. Rust, in particular, offers the choice of storing data on the stack or heap and determines at compile time when memory is no longer needed and can be cleaned up. This allows efficient use of memory as well as performant memory access.
Unlike many existing systems programming languages, Rust strives to have as many zero-cost abstractions as possible. To put it bluntly, the programmers' coding style should not affect the performance.
The biggest benefit Rust can provide compared to other languages is a borrow checker. This part of the compiler ensures that references, values that refer to specific data but do not own it, do not outlive this data. This provides the opportunity to eliminate entire classes of bugs caused by memory unsafety.
This benefit can also become a difficulty for the Rust programming language. The following example creates a mutable string that contains a name and then applies a reference to the first three bytes of the name. While that reference is outstanding, we attempt to mutate the string by clearing it. Since there is no guarantee that the reference points to valid data and the action of dereferencing could lead to undefined behavior, the compiler returns an error message:

```
fn no_mutable_aliasing() {
    let mut name = String::from("Vivian");
    let nickname = &name[..3];
    name.clear(); //Truncates this String, removing all contents.
    println!("Hello there, {}!", nickname);
}

error[E0502]: cannot borrow 'name' as mutable because it is also
borrowed as immutable
 --> a.rs:4:5
  |
```

[1] https://arewegameyet.rs
[2] for example Redox OS

```
3 | let nickname = &name[..3];
  | ---- immutable borrow occurs here
4 | name.clear();
  | ^^^^^^^^^^^^ mutable borrow occurs here
5 | println!("Hello␣there,␣{}!", nickname);
  | -------- immutable borrow
                                           later used here

For more information about this error, try 'rustc --explain E0502'.
```

Helpfully, the error message incorporates the code and tries hard to explain the problem, pointing out exact locations. There is a reference pointing to the string that the program has to clear. Doing so might require the string's memory to be freed, invalidating any existing references. If this happens and it is used in the value of "nickname" we would be accessing uninitialized memory, potentially causing a crash.

3 CDCL Sat Solver

Most current complete state-of-the-art SAT solvers are based on the Conflict Driven Clause Learning (CDCL) algorithm. Since their inception in the mid-90s, CDCL SAT solvers have been applied, in many cases with remarkable success, to several practical applications. Examples of applications include hardware and software model checking, planning, equivalence checking, bioinformatics, hardware, and software test pattern generation, software package dependencies, and cryptography [9]. This chapter surveys the organization of CDCL solvers.

3.1 DPLL

The concept of CDCL SAT solvers is primarily inspired by Davis–Putnam–Logemann–Loveland (DPLL) solvers. As a result, a reasonable knowledge of the organization of DPLL is assumed. In order to offer a detailed account of CDCL SAT solvers, several concepts have to be introduced, which serve to formalize the operations implemented by any DPLL SAT solver.
DPLL corresponds to backtrack search, where each step selects a variable and a propositional value for branching purposes. With each branching step, two values can be assigned to a variable, either 0 or 1. Branching corresponds to assigning the chosen value to the chosen variable. Afterward, the logical consequences of each branching step are evaluated. Each time an unsatisfied clause (i.e., a *conflict*) is identified, *backtracking* is executed. Backtracking corresponds to undoing branching steps until an unflipped branch is reached. Backtracking will undo this branching step when both values have been assigned to the selected variable at a branching step. If, for the first branching step, both values have been considered, and backtracking undoes this first branching step, then the CNF formula can be declared *unsatisfiable*. This kind of backtracking is called *chronological backtracking*. An alternative backtracking scheme is *non-chronological backtracking*, described later in this chapter [9].

3.2 Unit Propagation

In the last chapter, Boolean expressions were introduced. It was suggested that a Boolean expression is in CNF if it is a conjunction of one or more clauses, where a clause is a disjunction of variables.
Clauses are characterized as *unsatisfied*, *satisfied*, *unit* or *unresolved*. A clause is *unsatisfied* if all its literals are assigned value 0. A clause is *satisfied* if at least one of its literals is assigned value 1. A clause is *unit* if all literals but one are assigned value 0, and the remaining literal is unassigned. Finally, a clause is *unresolved* if it is neither *unsatisfied*, nor *satisfied*, nor *unit*.
A fundamental procedure in SAT solvers is the unit clause rule: if a clause is *unit*, then its

sole unassigned literal must be assigned value 1 for the clause to be satisfied. The iterated application of the unit clause rule is referred to as unit propagation. In modern CDCL solvers, as in most implementations of DPLL, logical consequences are derived with unit propagation. Unit propagation is applied after each branching step (and during preprocessing) and is used to identify variables that must be assigned a specific Boolean value. If an unsatisfied clause is identified, a conflict condition is declared, and the algorithm backtracks.
Finally, suppose a truth assignment is found for all or most variables in the original formula so that it makes all clauses *satisfiable*. In that case, the complete Boolean expression is said to be *satisfiable*. If that is not the case, the CNF formula is *unsatisfiable*.

3.3 Decision Level

In CDCL SAT solvers, each variable x_i is characterized by a number of properties, including the value the *antecedent* and the *decision level*, denoted respectively by $\nu(v_i) \in \{0, u, 1\}, \alpha(x_i) \in \phi \cup \{NIL\}$, and $\delta(x_i) \in \{-1, 0, 1, \ldots, |X|\}$. A variable x_i that is assigned a value as a result of applying the unit clause rule, is said to be implied. The unit clause ω used for implying variables x_i is said to be the antecedent of x_i, $\alpha(x_i) = \omega$. For variables that are decision variables or unassigned, the antecedent is NIL. Hence, antecedents are only defined for variables whose value is implied by other assignments. The decision level of a variable x_i denotes the depth of the decision tree at which the variable is assigned a value in $\{0, 1\}$. The decision level for an unassigned variable x_i is -1, $\delta(x_i) = -1$. The decision level associated with variables used for branching steps (i.e. decision assignments) is specified by the search process, and denotes the current depth of the decision stack. Hence, a variable x_i associated with a decision assignment is characterized by having $\alpha(x_i) = NIL$ and $\delta(x_i) > 0$. More formally, the decision level of x_i with antecedent ω is given by:

$$\delta(x_i) = max(\{0\} \cup \{\delta(x_j) | x_j \in \omega \wedge x_j \neq x_i\})$$

i.e. the decision level of an implied literal is either the highest decision level of the implied literals in a unit clause, or it is 0 in case the clause is unit. Moreover, the decision level of a literal is defined by the decision level as its variable, $\delta(l) = \delta(x_i)$ if $l = x_i$ or $l = \neg x_i$ [10].

3.4 CDCL

Most modern CDCL solvers do not use the simple DPLL backtracking search procedure approach. Instead, they use the Conflict Driven Clause Learning (CDCL) approach in which the idea is to iteratively decide a truth value to an unassigned variable and perform unit propagation 3.2 when a conflict (an unsatisfied clause) is obtained. After that, analyze the reason for the conflict, simplify it and learn a new clause that forbids this and possibly multiple similar conflict situations from occurring in the future, and backjump, possibly over several irrelevant decisions 3.3, based on the conflict analysis.
The two initial phases above are similar to the process of building branches in the simple DPLL search tree with unit propagation, but the last ones make the CDCL approach stronger

than DPLL. Because the learned conflict clauses have the unique property that they enable new unit propagations to occur after non-chronological backtracking, the search of CDCL is perceived as a sequence of transformations on ordered partial truth assignments.

A pseudocode for the CDCL algorithm is shown below.

def CDCL(ϕ): ▷ ϕ is a CNF formula
$\tau \leftarrow \emptyset$
while true: **do**
    $\tau \leftarrow unit - propagate(\phi, \tau)$ ▷ Unit propagation
    if τ falsifies a clause: **then**
        if at decision level 0: **then** return **unsat**
        end if
        $C \leftarrow analyze - conflict(\phi, \tau)$ ▷ Build the learned clause
        $\phi \leftarrow \phi \wedge C$ ▷ Add it to the formula
        backjump to an earlier decision level according to C
    else
        if all variables have values: **then** return **sat**
        end if
        start a new decision level
        choose a literal l such that $\tau(l)$ is undefined
        $\tau \leftarrow \tau \cup \{l\}$ ▷ "Decide" that l is true
    end if
end while

3.4.1 Restarts

Restarts are used in SAT solvers to avoid heavy-tail behavior [11]. Restart strategies have been a crucial feature in CDCL solvers to tackle hard industrial problems. The typical CDCL algorithm shown above does not account for a few often used techniques, namely search restarts and implementation of clause deletion policies. Search restarts, cause the algorithm to restart itself, but already learnt clauses are kept. Clause deletion policies are used to decide learnt clauses that can be deleted. Clause deletion allows the memory usage of a SAT solver to be kept under control.

3.4.2 Conflict Analysis

In this chapter, the procedure of conflict analysis will be explained in detail. As stated before, the conflict analysis process is invoked each time a solver encounters a conflict during unit propagation 3.2. Then the structure of unit propagation is analyzed, and it is decided which literals to include in the learnt clause.
Firstly, let us investigate how the solver finds a conflict by giving an example and constructing the implication graph. An *implication graph* is a directed acyclic graph, which helps demon-

strate the CDCL algorithm's functionality. In an implication graph, each vertex represents a variable assignment or, in other words, a literal. An incident edge to a vertex represents the reason leading to that assignment. These reasons are clauses that become unit and force the variable assignment.
For this reason, decision variables have no incident edges in contrast to implied variables with assignments during unit propagation. Each variable has a decision level 3.3 associated with it. If a graph contains a variable assigned both 0 and 1, that is, both x and $\neg x$ exist in the graph, then the implication graph contains a conflict.

Example 3.4.1. The steps taken in this example to represent the Conflict Driven Clause Learning procedure are the following:

1. Select a variable and assign True or False. This is called decision state. Remember the assignment.
2. Apply Boolean constraint propagation (unit propagation).
3. Build the implication graph.
4. If there is any conflict
   a) Find the cut in the implication graph that led to the conflict
   b) Derive a new clause which is the negation of the assignments that led to the conflict
   c) Non-chronologically backtrack ("back jump") to the appropriate decision level, where the first-assigned variable involved in the conflict was assigned
5. Otherwise continue from step 1 until all variable values are assigned.

Let us take the following set of clauses and start building the implication graph.

$$\begin{array}{rr} \{x_1 \vee x_4\} & \{x_2 \vee x_{11}\} \\ \{x_1 \vee x_8 \vee x_{12}\} & \{x_1 \vee \neg x_3 \vee \neg x_8\} \\ \{\neg x_7 \vee \neg x_3 \vee x_9\} & \{\neg x_7 \vee x_8 \vee \neg x_9\} \\ \{x_7 \vee x_8 \vee \neg x_{10}\} & \{x_7 \vee x_{10} \vee \neg x_{12}\} \end{array}$$

At first, we pick a branching variable, namely x_1, and take an arbitrary decision $x_1 = 0$. This happens at the decision level 1, as it is the first decision taken. Looking at the graph below 3.1, we can see that the variable x_1 is assigned the value 1 and the decision level is written on the right, by $d : 1$. Then we apply unit propagation, which yields that x_4 must be 1 (i.e., True). Particularly, this means that the clause $\{x_1 \vee x_4\}$ forces the variable x_4 to be 1, because x_1 was set to 1. Afterward, we arbitrarily pick another branching variable, $x_3 = 1$. The decision level is now 2. We apply unit propagation and find that the variables x_8 and x_{12} are forced to be assigned the values 0 and 1, respectively, because of the clauses $\{x_1 \vee \neg x_3 \vee \neg x_8\}$ and $\{x_1 \vee x_8 \vee x_{12}\}$. As we can see in the graph, the variables x_3, x_8, and x_{12} maintain the decision level 3. We then pick another branching variable, $x_2 = 1$, which leads to $x_{11} = 1$, at the decision level 3, from the clause $\{x_2 \vee x_{11}\}$. Later, we pick another

branching variable, $x_7 = 1$. The decision level is now 4. The variable x_9 is forced to be 1 from the clause $\{\neg x_7 \vee x_8 \vee \neg x_9\}$, as seen in the implication graph below. However, the clause $\{\neg x_7 \vee \neg x_3 \vee x_9\}$ forces the variable x_9 to be 0, so variable x_9 has the values 1 and 0 simultaneously. This means that a conflict is found!

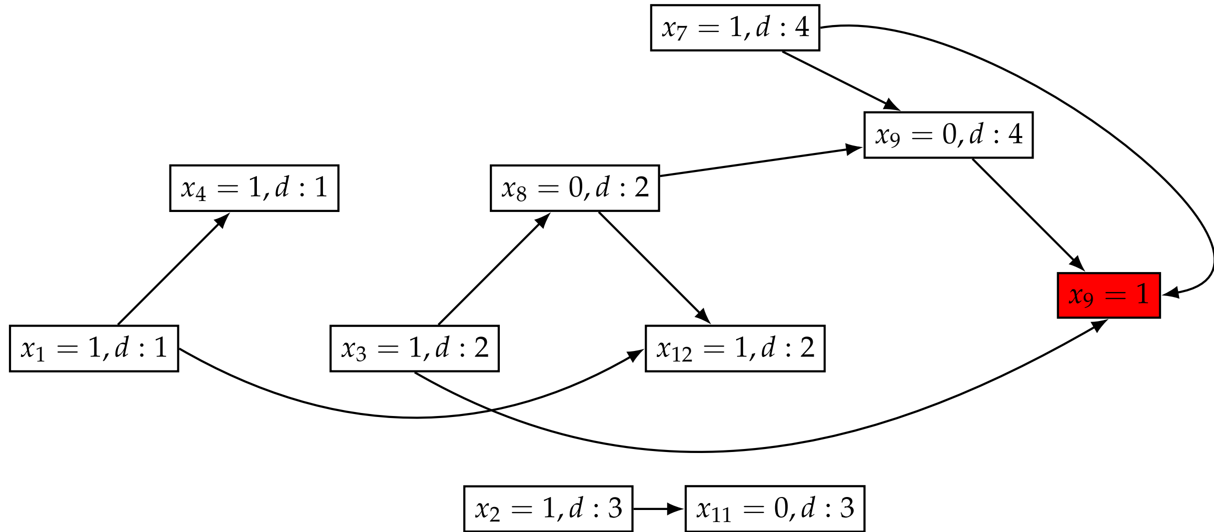


Figure 3.1: Implication Graph

Now, the cut that led to this conflict needs to be found. In case of a conflict, the implication graph can be split by a bipartition called a cut. The two sides of the partition can be referred to as the conflict side and the reason side of the implication graph. The conflict side contains the conflicting nodes. The reason side contains the nodes of the implication graph bipartition not included in the conflict. Various ways exist to create such extensions of the conflict side, equivalent to various cuts. Different cuts of the implication graph distinguish learning schemes from one another because the conflict clause, and thus the knowledge gained from the conflict, is derived from the bipartition of the implication graph.

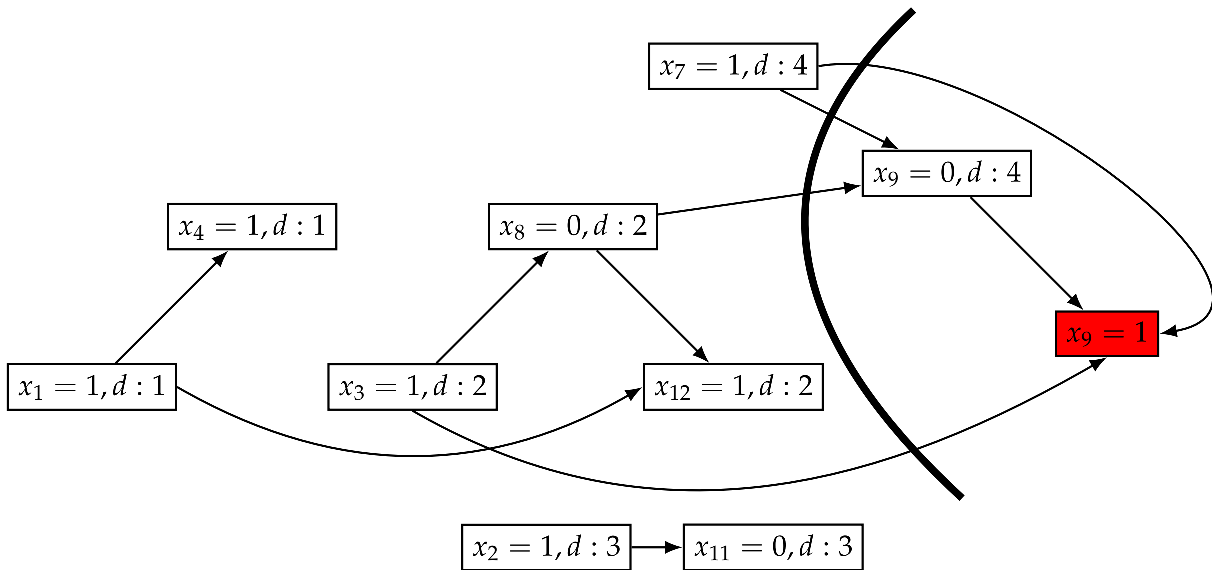


Figure 3.2: Implication Graph

The cut is drawn in figure 3.2. From the cut, we find a conflicting condition, which is $x_3 = 1 \vee x_7 = 1 \vee x_8 = 0 \rightarrow conflict$. We take the negation of this condition and make it a clause, $\neg x_3 \vee \neg x_7 \vee x_8$. This is not the only possible cut; a cut after the variable x_9 would also be possible, and the learnt clause would then be $\{\neg x_7 \vee \neg x_3 \vee x_9\}$. As this clause already exists in the initial clause set, we take the one mentioned above. Finally, we add the conflict clause to the problem and apply a non-chronological back jump to the appropriate decision level, which in this case is the second highest decision level of the literals in the learned clause, meaning to the decision level 2 of $x_3 = 1, d : 2$ with the condition $x_7 = 0$. After the Back jump, the variable values until decision level 2 are set accordingly, and the ones in a higher decision level are erased, as seen in the graph 3.3. We continue then the procedure.

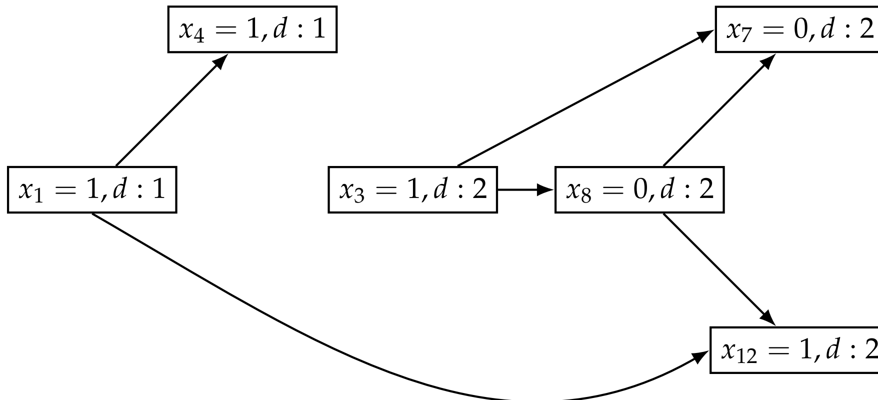


Figure 3.3: Implication Graph

3.4.3 Clause Learning

As shown in the above example, when CDCL reaches a conflict, it analyzes the arbitrary decisions 3.3 made and all of the assignments forced to infer via unit propagation 3.2, eventually leading to the conflict. CDCL is now able to learn a clause that is potentially more useful than just the knowledge that the current partial assignment was poor. This way, CDCL can avoid making the same mistake over and over and skip over large chunks of bad partial assignments DPLL will get stuck in.
Furthermore, a learned clause allows CDCL to learn from the decisions made and ignore vast sections of the search space that will never satisfy the formula.
Once CDCL has learned a clause, the algorithm can backtrack potentially more than one level, and unit propagate using the learned clause to put what it learned into action immediately. The ability to backtrack more than one level with the learned clause is called *non-chronological backtracking*.
Finally, by repeating the process, CDCL is reaching a final assignment for the variables of the initial formula faster and easier.

As a conclusion to this chapter, it is significant to point out that CDCL is a sound and complete algorithm for the Boolean Satisfiability Problem. To certify the result of a CDCL SAT solver, two cases have to be considered:

- SAT: The algorithm returns a satisfying assignment for the formula.
- UNSAT: The sequence of learned clauses is a certificate of unsatisfiability.

4 Parallel BDD Library in Rust to support the CDCL process

In this chapter, the BDD library implemented in this Master thesis project will be presented in detail by investigating the key ideas and demonstrating the more essential algorithms. Notably, the existing BDD libraries will be briefly introduced. Additionally, it will be researched how the BDD is created and which algorithms are used. Lastly, it will be investigated how the BDD is being approximated throughout the construction process and how the witness clauses are generated.

4.1 Existing BDD Libraries

Currently, several libraries exist to generate BDDs from Boolean functions, the most widespread being BuDDy and CUDD, both of which are not maintained anymore by their creators. Both BuDDy and CUDD require expert knowledge about BDDs to use and do not support parallelized execution. There exist libraries for parallelized BDD execution, like Sylvan, but they are less potent than BuDDy and CUDD.

4.1.1 CUDD

CUDD stands for Colorado University Decision Diagram, and it was written by Fabio Somenzi. This library cannot only construct BDDs, but it can also create and calculate other types of decision diagrams. CUDD supports the complete Bryant API [5] to manipulate BDDs. It was released in 1995, which means it is now close to 30 years in use. Therefore it is thoroughly tested. CUDD is a manager-based package, and it supports dynamic variable ordering 4.3.2. Geert Janssen ranked CUDD as one of the eight good packages they researched. CUDD was also not planned with parallelization in mind while creating the library.

4.1.2 BuDDy

BuDDy is a BDD package written by Jørn Lind-Nielsen; it supports the complete Bryant API [5] for BDD manipulation. BuDDy also supports dynamic variable ordering 4.3.2. BuDDy is written in the C programming language but wrapped in a C++ layer to make it more accessible. BuDDy is not manager based, which means it can only calculate one BDD in a single program execution. BuDDy also implements some sophisticated drawing functions, useful for debugging. It is possible to send the BDD output to the graph printing program

DOT1 to draw created BDDs. Geert Janssen ranked CUDD as one of the eight good packages they researched. BuDDy supports parallel execution.

4.1.3 Sylvan

Sylvan is a BDD package written by Tom van Djik which was designed with parallelization in mind, although it does not support dynamic variable ordering 4.3.2. It implements the Bryant API [5], but in a way that supports parallel execution of these operations. Sylvan supports more decision diagrams than just BDDs. Sylvan is written in C but also provides a C++ layer, plus there are bindings for Java, Haskell, and Python. Sylvan is actively developed and maintained. Unfortunately, Sylvan was not ranked by Geert Janssen. However, Sylvan was ranked by Tom van Djik et al., and they showed that parallelized BDD construction could outperform sequential construction if the circumstances are right (overall long construction times where multiple CPU cores can be used).

4.2 BDD Library Design

This thesis addresses the shortcomings by providing an architecture for a BDD library with parallelism as an essential characteristic and a modular architecture. The different algorithms used for constructing the BDD can be exchanged to measure the runtime impacts of different implementations. Particularly, the lack of parallelization in existing BDD libraries was not only resolved by enabling the new BDD library to run multiple instances of itself or run as another thread parallel to another SAT solver. This BDD library sends the learnt clause to the other thread(s) immediately after the witness clause is found and is also receiving instantly a learnt clause to its vector of clauses to be processed. As a result, a proof-of-concept implementation in the Rust programming language is provided, which can construct BDDs using parallelism, solve SAT problems and provide manipulation through Boolean functions.

Designing a software architecture requires careful planning to provide all the requirements specified beforehand. This library's final design must be extensible, maintainable, and parallelizable. The architecture structure is outlined in Figure 4.1. The first step is reading the input file in cnf format and passing it to the parser to transform it into a Dimacs format. The DIMACS CNF format is a textual representation of a formula in conjunctive normal form. Afterward, a variable ordering 4.3.2 algorithm is applied to the intermediate representation. In the last steps, while the BDD is being constructed, the learnt clauses generated are sent to Glucose through the Clause Database 4.3.3, the BDD is being approximated, and a Message Parsing Interface (MPI) is active to identify notifications like for example if Glucose has terminated.

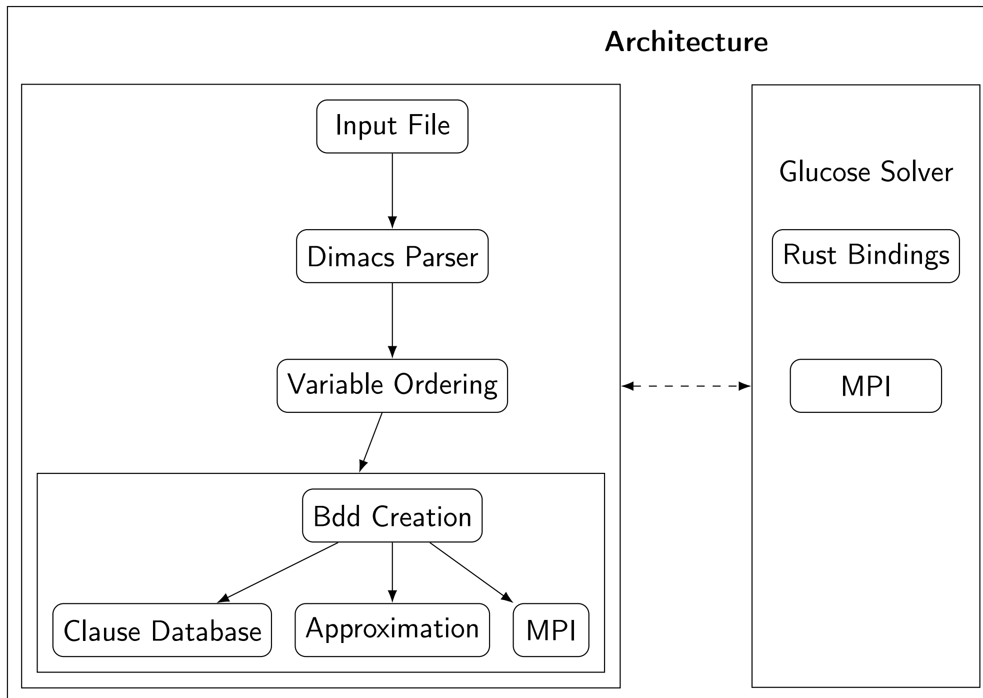


Figure 4.1: Bdd Library Architecture

4.3 Modular Components

In this section, all modular components of the architecture will be discussed, and their tasks explained. As described above and displayed in Figure 4.1, the architecture contains three modular stages to arrive at the overall management of the creation of the parallel BDD. The Glucose SAT Solver is connected to the BDD architecture through bindings and an MPI interface to enable communication between the two architectures. The connection between BDD and Glucose architectures will be discussed in detail in the next chapter.

4.3.1 DIMACS Parser

First, the parser receives a path to a cnf file and reads it. Then it uses the PEST grammar to translate the String input into a vector of Boolean Expressions. Pest is a general-purpose parser written in Rust with a focus on accessibility, correctness, and performance. It uses parsing expression grammars (or PEG) as input, which are similar in spirit to regular expressions but offer the enhanced expressivity needed to parse complex languages.
The intermediate representation of Boolean Expressions is defined in the following code sample.

```
/// Recursive implementation of boolean expression.
/// Firstly only the ones important for SAT Solving clauses
/// will be considered. It can be extended afterwards.
```

```
#[derive(Clone, Debug, Eq)]
pub enum Expression {
    Constant(bool),
    Variable(i32),
    Not(Box<Expression>),
    Or(Box<Expression>, Box<Expression>),
    And(Box<Expression>, Box<Expression>),
}
```

A Boolean Function, or in this example *Expression*, consists either of a *Constant*, a *Variable*, or an *Operator*. The *Operator* represents the negation, the logical conjunction, or the logical disjunction operation. In the case of conjunction or disjunction, it has a left-hand side and a right-hand side which are by themselves of type $Box < Expression >$. The recursive expressions are wrapped in a *Box* because the size of the *Expression* is not known at compile time. Hence, the memory on the heap has to be allocated. This format allows building any Boolean function by recursion.

4.3.2 Variable Ordering

After the input format was passed through the parser, the variable ordering algorithm was applied. As mentioned before, a beneficial variable ordering can significantly reduce the size of the BDD. The above will be proven with the help of the following theorems derived from the paper "BDD-Guided Clause Generation" [1].

Definition 4.3.0.1. A *witness clause* is a clause deduced from the infeasible nodes of the BDD (that is, the nodes from which no path leads to the true sink) by using the state information for these nodes.

Theorem 4.3.1. Let F be a CNF formula, and B be a top-down exact or restricted BDD for F constructed as described above. Then:

1. Every witness clause generated from B is valid for F.
2. The set of variables in every witness clause generated at layer L_{i+1} is a subset of $x_1, x_2, ..., x_i$.
3. The witness clause C generated for a node v of B is falsified by the partial assignment corresponding to every path from the root of B to v. In particular, C does not contain any variable that appears both negatively and positively in paths from the root to v.
4. The witness clause C associated with any infeasible node v of B witnesses the infeasibility of v.
5. Let U denote the roots of maximal infeasible subtrees of B. Then the set G of all witness clauses associated with nodes $v \in U$ is a reformulation of F.

Theorem 4.3.2. For any clause C learned from one application of SAT conflict analysis on a clause set F using any clause learning scheme, there exists a variable ordering under which a top-down approximate BDD of width at most $2^{|C|}$ for F generates a clause $C' \subseteq C$.

Proof. To prove this, we use the resolution-based characterization of CDCL clauses, namely, the CDCL derivation of a clause C. Starting from F and using any clause learning scheme, it can be viewed as a simple form of resolution derivation with a ladder-like structure. More formally, the derivation τ of C is simultaneously a tree-like, regular, linear, and ordered resolution derivation from the clauses in F. This means that each intermediate clause C_{j+1} in τ is obtained by resolving C_j with a clause of F. Additionally, it suggests that the sequence σ of variables resolved upon in τ consists of all distinct variables.
We can use BDDs to derive from F a clause C' that, together with F, absorbs C. To construct such a BDD B, we use as the top-down (partial) variable order first the variables that appear in C (in any order) followed by variables in the reverse order of σ. The first $|C|$ variables result in a BDDs of width at most $2^{|C|}$. Let v be the node of B in the layer $L_{|C|+1}$ at which all literals of C are falsified. When expanding B from v, the ladder-like structure of τ guarantees the following. At least one branch on the variables in σ can be labeled directly by a clause of F that is falsified. The corresponding lower part of B starting at v is thus of width 1. For the remaining $2^{|C|} - 1$ nodes of B in the layer $L_{|C|+1}$, we construct an approximate lower portion of the BDD such that the overall width does not increase. This makes the overall width of B $2^{|C|}$.
While B may have several infeasible nodes, the node v in the layer $L_{|C|+1}$ is guaranteed by the derivation τ to be infeasible. Recall that the path p from the root of B to v falsifies C. Consider the node v' that is the root of the maximal infeasible subtree of B that contains v. Let C' be the BDD-generated clause witnessing the infeasibility of v'. By Theorem 4.3.1, C' must be falsified by the path p' from the root of B to v'. Note that p' is a sub-path of p. By construction, C contains all $|C|$ literals mentioned along p, while, by Theorem 4.3.1, C' contains a subset of the literals mentioned along p' and hence along p. It follows that $C' \subseteq C$. □

In the implementation of this Masters project, a simple heuristic to determine the variable ordering is used: each variable is assigned a score, computed as the quotient between the number of clauses containing the variable and the average arity of those clauses, and the variables are sorted in decreasing order according to this score so that higher-scoring variables (that is, variables that appear in primarily many short clauses) correspond to layers nearer the top of the BDD. The idea for the above variable ordering came from the paper BDD-Guided Clause Generation [1].

4.3.3 BDD Creation

Before the BDD construction begins, the Clause Database is initialized.

Clause Database

The Clause Database is a data structure whose purpose is to manage the clause exchange between BDD and Glucose. The Clause Database acts as a sharing manager between Glucose and BDD, as it contains different filters, which help manage the information exchange. The clauses sent from both architectures are received by the Clause Database, filtered using Bloom filters 4.3.3, and sent back to the architectures.

Bloom filters

Bloom filters are used to detect duplicate clauses. A Bloom filter is a space-efficient probabilistic set data structure that allows false-positive matches, meaning that some clauses might be considered duplicates even if they are not. The library *BloomFilters* in Rust is used for the Bloom filter implementation. Bloom Filters are helpful for situations where the size of the data set is not known ahead of time. For example, a Stable Bloom Filter can deduplicate events from an unbounded event stream with a specified upper bound on false positives and minimal false negatives. A classic Bloom filter used in this implementation is a particular case of a Stable Bloom Filter whose eviction rate is zero and cell size is one. More on Bloom filters can be found in the paper Space/time trade-offs in hash coding with allowable errors [12].

After the Clause Database 4.3.3 is initialized, the actual parallel creation of the BDD starts. In chapter 2 it was explained how an OBDD is constructed from a Boolean Expression using the Shannon expansion after a variable ordering 4.3.2 is set. It was then stated that an OBDD could be reduced and become a ROBDD after it is constructed. In this implementation, the OBDD is being reduced while it is being built. The needed algorithms for creating the ROBDD will be researched in this section.
Starting from a Boolean Expression t the variable ordering $\{x_i, x_{i+1}, ..., x_n\}$ based on each variables score is constructed. The nodes map table T is initiated, and the *zero* and *one* terminal nodes are added.
In the beginning, the following algorithm $BUILD$ to t.

```
/// Construct a Robdd from a given expression
pub fn build(&self, expr: &Expr) -> Bdd {
    match expr {
        Const(value) => Bdd::new_value(BddVar::new(i32::MAX), value),
        Var(name) => {
            let var = BddVar::new(*name);
            Bdd::new_var(var)
        },
        Not(inner) => self.build(inner).negate(),
        And(l, r) => {
            let (left,right) = rayon::join(|| self.build(l), || self.build(r));
            self.and(&left, &right)
        },
```

```
        Or(l, r) => {
            let (left,right) = rayon::join(|| self.build(l), || self.build(r));
            self.or(&left, &right)
        }
    }
}
```

The Boolean expression is matched, and in the case of a constant, a new terminal BDD is produced depending on the constant's value. In the case of a variable, a new non-terminal BDD is created, connecting the variable to the low pointer *zero* and high pointer *one*. In the case of negation, the BDD for the inner subexpression is recursively created and then negated. In the case of conjunction or disjunction, the two subtrees from the inner subexpressions are constructed recursively. Then the binary operator is applied with the help of the algorithm *APPLY* stated below.

APPLY[T,H](op, u_1, u_2)
$init(G)$
function A(P)P(u_1, u_2)
  if $G(u_1, u_2) \neq empty$ **then**
    return $G(u_1, u_2)$
  else if $u_1 \in \{0,1\}$ and $u_2 \in \{0,1\}$ **then**
    $u \leftarrow op(u_1, u_2)$
  else if $var(u_1) = var(u_2)$ **then**
    $u \leftarrow MK(var(u_1), APP(low(u_1), low(u_2)), APP(high(u_1), high(u_2)))$
  else if $var(u_1) < var(u_2)$ **then**
    $u \leftarrow MK(var(u_1), APP(low(u_1), u_2), APP(high(u_1), u_2))$
  else$var(u_1) > var(u_2)$
    $u \leftarrow MK(var(u_1), APP(u_1, low(u_2)), APP(u_1, high(u_2)))$
  end if
  $G(u_1, u_2) \leftarrow u$
  return u
end function

The algorithm *APPLY* takes two BDDs formed in the *BUILD* method and a binary operator. Each BDD is a vector of nodes, and the nodes map T holds the pointers connected to these nodes.
Firstly, starting from the root of both BDDs, root pointers to the root nodes, respectively, are considered. Precisely, a node consists of a variable (*var*), a pointer to the low child (*low*), and a pointer to the high child (*high*) of the node. Taking two nodes v_1 and v_2, it is determined whether var_1 is equal, greater, or smaller than var_2 depending on each variable's score. The variables with the lowest score will be the root variable of the resulting node.
Afterwards, the final node is created by a call to the *mk* algorithm presented below. In order to ensure that the OBDD being constructed is reduced, it is necessary to determine from a

triple (i, l, h) whether there exists a node v with $var(v) = i$, $low(v) = l$ and $high(v) = h$. For this purpose, we assume the presence of a table $H : (i, l, h) \rightarrow v$ mapping triples (i, h, l) of variables indices i and nodes l and h to v. If no such node exists, the node is created, inserted into the table H, and a pointer to it is returned. The table H is the "inverse" of the table T of variable nodes v. It holds that $T(v) = (i, l, h)$ if and only if $H(i, l, h) = v$. The runtime of the *mk* algorithm is $O(1)$. The node is then added to the resulting BDD.

MK[T,H](i,l,h)
if $l = h$ **then** return l
else if $member(H, i, l, h)$ **then**
    return $lookup(H, i, l, h)$
else $v \leftarrow add(T, i, l, h)$
    $insert(H, i, l, h, v)$
    return v
end if

Finally, the rest of the nodes are constructed and recursively added to the BDD. It is important to note here a fundamental characteristic of this implementation that makes it effective. From the above algorithms, it can be observed that $BUILD$ deconstructs the expressions from the inner subexpression, starting from *zero* and *one*, so bottom-up and $APPLY$ constructs the resulting BDD starting from the roots of the sub-BDDs, so top-down.

Another significant characteristic of this implementation is that it considers the initial Boolean expression t as a vector of pairs of subexpressions. In detail, as t resulted from a CNF file, it can be seen as a vector of disjunctions. That enables the implementation to construct temporary BDDs for each disjunction pair and then use the algorithm $APPLY$ again to connect the temporary BDD with the current BDD with a conjunction. This property allows for the witness clauses to be identified and sent from the current BDD at the same time as the new temporary BDD is constructed. This is achieved with the help of a rust crate named Rayon. Rayon is a data-parallelism library for Rust. It is incredibly lightweight and makes it easy to convert a sequential computation into a parallel one. It also guarantees data-race freedom.

Witness Clause Generation

During the top-down construction of a BDD for a SAT instance, the infeasibility of a state is detected when an unsatisfied clause contains no variable corresponding to a lower layer of the BDD. When this occurs, we choose one such clause as a witness of the infeasibility of the corresponding node.

Example 4.3.1. Let us consider the BDD with the variable ordering $x_1 < y_1 < x_2 < y_2$ as seen in figure 4.2. First, we identify the *zero* node. Now, in a bottom-up pass, we construct the infeasible paths, which are $\neg x_1 \vee \neg y_1$, $x_1 \vee \neg y_1 \vee \neg y_2$ and $\neg x_1 \vee y_1 \vee \neg x_2 \vee y_2$. The witness clauses are then constructed by negating each infeasible path, and we get $x_1 \vee y_1$, $\neg x_1 \vee y_1 \vee y_2$

and $x_1 \vee \neg y_1 \vee x_2 \vee \neg y_2$. These clauses will be sent to the Clause Database 4.3.3, filtered through Bloom filters 4.3.3, and passed to Glucose.

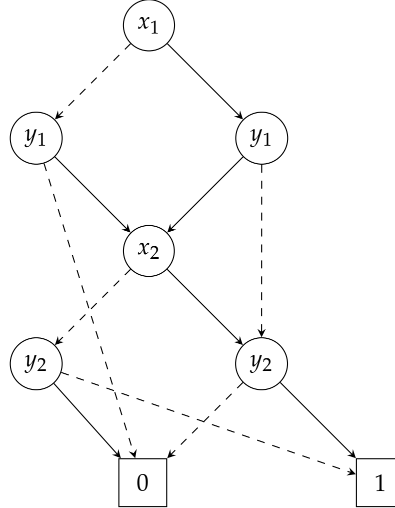


Figure 4.2: An Reduced Ordered Binary Decision Diagram with Variable Ordering $x_1 < y_1 < x_2 < y_2$

Let us formally define a clause C to be valid for a propositional formula F if $F| = C$, i.e., F logically entails C. In other words, $F \wedge C$ has the exact solutions as F. Initially, theorem 4.3.1 from the paper "BDD-Guided Clause Generation" [1], which was first viewed in the above section of variable ordering 4.3.2, will now be presented again and proved in order to investigate more profound the concept of witness clause and understand why they are beneficial for the solver itself and for other solvers running in parallel.

Definition 4.3.2.1. A *state* is a partial assignment, which is as well referred to as a path in this thesis. A state function for layer i is a map σ_i from the set $\{0, 1\}^{i-1}$ of partial assignments at layer i into some set S_i of states, such that $\sigma_i(y) = \sigma_i(y')$ implies $S(y) = S(y')$. In other words, two partial assignments leading to the same state have the same satisfying completions.

Theorem 4.3.3. Let F be a CNF formula and B be a top-down exact, or restricted BDD for F constructed as described above. Then:

1. Every witness clause generated from B is valid for F.
2. The set of variables in every witness clause generated at layer L_{i+1} is a subset of $\{x_1, x_2, ..., x_i\}$.
3. The witness clause C generated for a node v of B is falsified by the partial assignment corresponding to every path from the root of B to v. In particular, C does not contain any variable that appears both negatively and positively in paths from the root to v.
4. The witness clause C associated with any infeasible node v of B witnesses the infeasibility of v.

5. Let U denote the roots of maximal infeasible subtrees of B. The set G of all witness clauses associated with nodes $v \in U$ is a reformulation of F.

Proof. The first claim follows immediately from the observation that the witness clause C associated with any node v is derived using a sequence of resolution operations starting from the clauses in $state(v)$. Since resolution is a sound proof system, $state(v)$, and hence F, must entail C.
We prove the second claim by induction on i. For $i = n + 1$, the claim trivially holds. Suppose the claim holds for clauses generated at layer L_i, with $i > 1$. By construction, any clause C generated at layer L_{i-1} either is identical to a clause generated at layer L_i, in which case it does not contain the variable x_{i-1}, or else is obtained by resolving two clauses at layer L_i on the variable x_i. In either case, by the induction hypothesis, the variables appearing in C must be a subset of $\{x_1, x_2, ..., x_{i-2}\}$.
To prove the third claim, we recall from the definition of the state function that the partial assignment y corresponding to any path from the root to v does not satisfy any clause in $state(v) = F_v \subseteq F$. For the sake of contradiction, suppose l is a literal of the witness clause C that is satisfied by y. Since C is derived by applying resolution steps to clauses in F_v, the literal l must appear in at least one clause C' of F_v. Since y satisfies l, it would also satisfy C', a contradiction. Hence, C must be falsified by y. Finally, if C contained a literal l that appears positively and negatively in partial assignments y and y' corresponding to two paths from the root to v, then C would be satisfied by at least one of y and y', which, as proved above, cannot happen. Hence, C must not contain any such literal.
For proving the fourth claim, we use the above property that the partial assignment y corresponding to any path from the root to v does not satisfy C. Suppose y could be extended to a full assignment (y, z) that satisfies F. Then z must satisfy all clauses in $state(v) = F_v$ as these clauses, by definition of the state function, are not satisfied by y. Since C is derived from F_v by applying a sequence of resolution operations, z must satisfy C. However, as observed above, C is a subset of $\{x_1, x_2, ..., x_{i-1}\}$, where L_i is the layer containing v, and hence C cannot possibly be satisfied by z. This proves that y cannot be extended to a full assignment satisfying F and that the generated clause C witnesses this fact, as well as the infeasibility of v.
Lastly, we argue that the set G of witness clauses associated with roots of maximal infeasible subtrees of B is logically equivalent to F. If y is a solution to F, then y must satisfy all witness clauses as these clauses are entailed by F. Hence y must also satisfy G. On the other hand, if y is not a solution to F, then let y' be the partial assignment corresponding to the path in B associated with y but truncated at the root v' of a maximal infeasible subtree. By the third property above, y' (and hence y) must falsify the clause C' associated with v', and hence falsify G. It follows that F and G have the same set of solutions, and thus G is a reformulation of F. □

The witness clauses generated from the BDD and sent over to Glucose also support the CDCL process and make it more effective. To make this connection explicit, the notion of absorbed clauses is recalled. A clause C is said to be absorbed by a CNF formula F, if for

every literal $l \in C$, performing unit propagation on F, starting with all literals of C except l set to false, the clause either infers l or infers a conflict. The intuition here is that C is absorbed by F if F and $F \wedge C$ have identical entailment power concerning unit propagation, i.e., whatever one can derive from $F \wedge C$ using unit propagation, one can also derive from F itself. Pipatsrisawat and Darwiche, in their paper "On the power of clause-learning SAT solvers as resolution engines" [13] showed that the CDCL mechanism in SAT solvers always produces clauses that are not absorbed by the current theory, that is, by the set of initial clauses of F, and those learned thus far during the search. This property also holds for clauses generated by the BDD method applied to F. The proof can be found in Sabharwals', Kenns', and van Hoeves' paper "BDD-Guided Clause Generation" [1].

Proposition 4.3.3.1. There exist BDD-generated clauses that cannot be derived using one application of SAT conflict analysis.

Example 4.3.2. Let us take the following clauses as in chapter 3. By building the implication graph, we resulted in a conflict and found the learnt clause: $\neg x_3 \vee \neg x_7 \vee x_8$.

$$
\begin{array}{rr}
\{x_1 \vee x_4\} & \{x_2 \vee x_{11}\} \\
\{x_1 \vee x_8 \vee x_{12}\} & \{x_1 \vee \neg x_3 \vee \neg x_8\} \\
\{\neg x_7 \vee \neg x_3 \vee x_9\} & \{\neg x_7 \vee x_8 \vee \neg x_9\} \\
\{x_7 \vee x_8 \vee \neg x_{10}\} & \{x_7 \vee x_{10} \vee \neg x_{12}\}
\end{array}
$$

Let us now take the variable ordering $\{x_1 < ... < x_{12}\}$ 4.3.2 and start building the ROBDD.

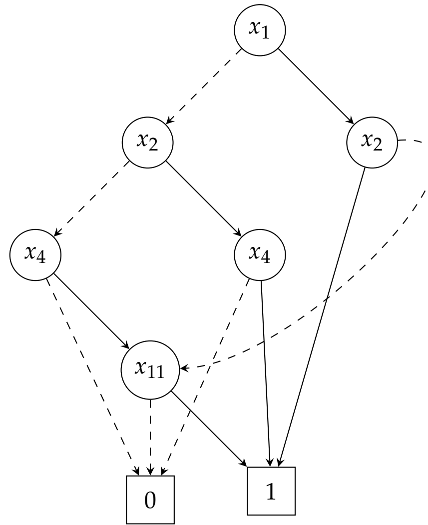


Figure 4.3: An Reduced Ordered Binary Decision Diagram from clauses $\{x_1 \vee x_4\}$ and $\{x_2 \vee x_{11}\}$.

The infeasible path $\neg x_1 \vee \neg x_2 \vee x_4 \vee \neg x_{11}$ is found and the witness clause $x_1 \vee x_2 \vee \neg x_4 \vee x_{11}$ is generated. If we add the clause to the initial set of clauses, generate the implication graph, and apply unit propagation, this does not lead to a conflict. The above proves that at least one clause is generated from the BDD methods that cannot be derived using SAT conflict analysis.

In chapter 6, based on the test results, it will be proven that the clauses sent from the BDD to Glucose are not absorbed by the set of initial clauses F, and most of them cannot be derived from the CDCL conflict analysis. As a result, the BDD library implemented in this Master's thesis is indeed supporting the CDCL SAT solver and making the whole process more effective.

Clauses are not only sent from the BDD to Glucose but also from Glucose back to the BDD. The purpose of this is to support the BDD as well as to generate stronger clauses and expedite the construction process. The communication between BDD and Glucose will be presented in detail in the next chapter.

Approximation

Whereas in this Masters project, the OBDDs created are restricted (ROBDDs), they can still grow exponentially, which decelerates the search operation. A solution to this problem is to approximate the ROBDDs while being constructed. Such BDDs are called approximate BDDs because their structure approximates the structure of the exact or restricted BDD. There are several approximation methods; for example, the creation of MDDs of limited width was proposed by Andersen et al. [14] to reduce space requirements. In this approach, the MDD is constructed top-down, layer-by-layer. Whenever a layer of the MDD exceeds some predetermined value W, an approximation operation is applied to reduce its size to W before constructing the next layer.
The approximation method used in this implementation is rounding inspired and firstly presented in the paper "Error Bounded Exact BDD Minimization in Approximate Computing" [15]. For a BDD node, we denote the *one*-set as the set of paths to *one*, and analogously the *zero*-set as the set of paths to *zero*. In rounding, for all nodes above a certain variable (in variable order of the BDD), the child with the smaller *one*-set is replaced with *one* or *zero*. Replacing with *one* is known as rounding up, and replacing with *zero* is known as rounding down. We can also guarantee that the algorithm terminates by gradually changing the boundary. Rounding up is a positive approximation since only satisfying assignments are added. As described in the paper, this technique gives good results. A variation of rounding up is therefore also used in the solvers. Instead of choosing the child with the smaller *one*-set, the child with the smaller *zero*-set is selected. This has the motivation that as few non-satisfying assignments as possible should be lost.

Example 4.3.3. Figure 4.4 shows the ROBDD before rounding up from variable x_6 and figure 4.5 shows the ROBDD before rounding up from variable x_6. The ROBDD shown in figure 4.5 approximated version of the one in figure 4.4. When approximating, only the left child of the node of x_6 is replaced by *one*. However, this has the consequence that some levels above it collapse, and the ROBDD thus becomes significantly smaller:

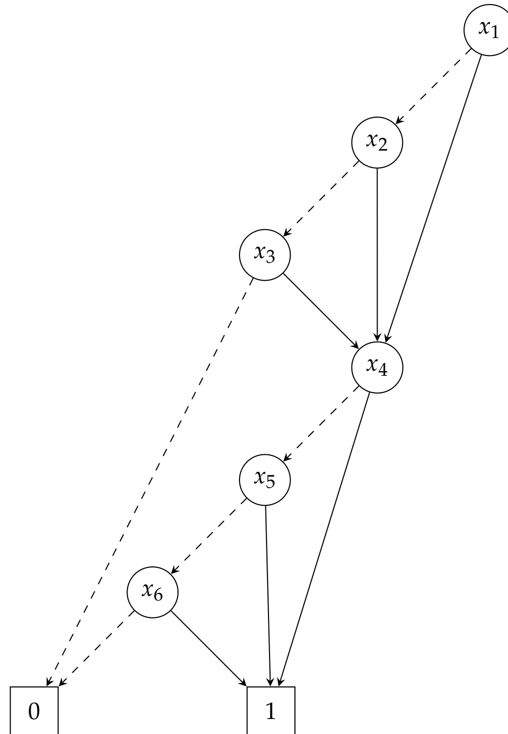


Figure 4.4: An Reduced Ordered Binary Decision Diagram before rounding.

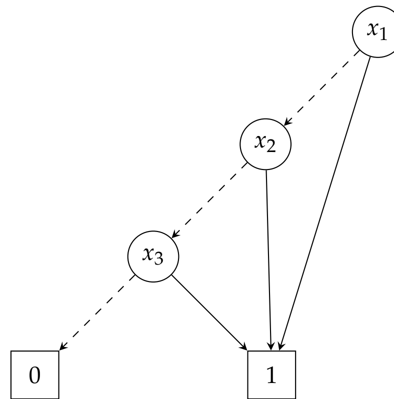


Figure 4.5: An Reduced Ordered Binary Decision Diagram after rounding.

Solution

The BDD library implemented is not only implemented to support a CDCL solver but can be used as a BDD solver on its own and find a solution for a CNF formula. As suggested in the paper "Integrating CNF and BDD Based SAT Solvers" [16], firstly, the terminal node *one* is located if it exists. Later, the paths from the terminal node to the root of the BDD are extracted by a bottom-up search heuristic. These are the solution sets. Before delivering the

solution, each element of the solution set is verified as satisfiable. Finally, the satisfiable path is returned as the adequate solution to the initial CNF formula.

Looking at the BDD from the last example, the assignment $\{x_1 = 0, x_2 = 0, x_4 = 1, x_{11} = 1\}$ can be obtained as a solution by following the above mentioned heuristic.

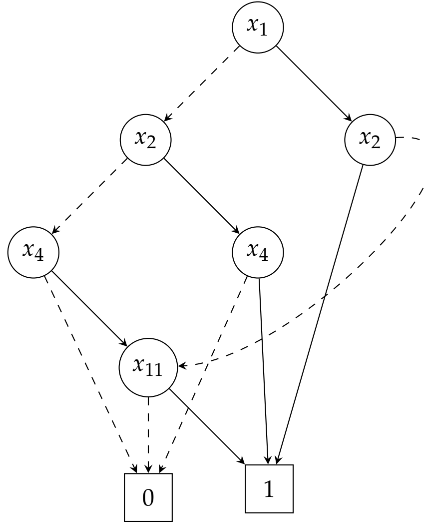


Figure 4.6: An Reduced Ordered Binary Decision Diagram from clauses $\{x_1 \vee x_4\}$ and $\{x_2 \vee x_{11}\}$.

Indeed, if we substitute the variables in the initial set with the assignments found in the solution set $\{x_1 = 0, x_2 = 0, x_4 = 1, x_{11} = 1\}$, all clauses are satisfiable. Have in mind, that the BDD pictured is a part-BDD built from the clauses

$$\{x_1 \vee x_4\} \wedge \{x_2 \vee x_{11}\}.$$

Summarising, all of the above is why a BDD library had to be implemented and one already existing could not be used. The core ideas and algorithms behind this implementation provoke the results presented in the following chapters.

5 Communication between the architectures

In this chapter, the information exchange between BDD and the SAT solver used in this project, Glucose, will be further investigated, and detailed examples will be given to explain the concept further. At first, essential information of the Glucose SAT solver is presented.

5.1 Glucose

Glucose is based on a scoring scheme introduced in 2009 for the clause learning mechanism of so-called "Modern" SAT solvers [17]. It has been designed to be parallel since 2014 and was entirely rebooted in 2021. Gilles Audemard and Laurent Simon have coded and maintained Glucose since its beginning.
The name of the Solver name is a contraction of the concept of "glue clauses," a particular kind of clauses that Glucose detects and preserves during the search. Glucose is heavily based on Minisat and has won nine awards in the latest SAT competitions.
Learning (in CDCL algorithms) was firstly introduced for completeness. Nevertheless, if all of Glucose 2's traces of the competition 2011 are studied, for instance, phase 2, in the categories Applications and Crafted, Glucose 2 learnt 973,468,489 clauses (sum over all traces) but removed 909,123,525 of them, i.e., more than 93% of the clauses are removed. This view is new and contradicts previous beliefs. Thus, it is believed by the authors of Glucose that one of the performance keys of the solver is not only based on the identification of good clauses. It is also a result of the removal of bad ones. As a side effect, by aggressively deleting those clauses, Glucose increases the CDCL incompleteness (keeping learnt clauses is essential for completeness). It should also be emphasized here that Glucose 2 was ranked fourth on the parallel track in the SAT 2011 competition, besides being sequential. This shows that even with a single core, the solver performed better in user time (not CPU) than many parallel solvers exploiting the eight cores of the parallel machine. One of the reasons for this is that Glucose 2 is good at finding the shortest (but easiest) proof possible.
Glucose 4.2.1 2018 is the last official release of Glucose. The same unmodified version was then used for a few competitions. This is the release of Glucose that was first used for the SAT Competition 2018 and then for SAT-Race 2019. The main modifications are based on the extension of the LCM strategies proposed at the International Joint Conference on Artificial Intelligence 2017, which" revived" the vivification technique).

5.2 Data Parallelism

The difference between multithreading and data parallelism is that in the first case, the CPU switches between different threads fast, giving a falsehood of concurrency. Keypoint is that only one thread is running at any given time. When one thread is running, others are blocked. In the second case, threads run parallel, usually in different CPU cores, ensuring true concurrency. Keypoint is that multiple threads are running at any given time. It is valuable for heavy computations and super long-running processes.
In this Master's project, data parallelism was used instead of multithreading to achieve the best results. Rust provides a data parallelism library named Rayon, specifically a method called *join*. By default, Rayon uses the same number of threads as the number of CPUs available. Note that on systems with hyperthreading enabled, this equals the number of logical cores, not the physical ones. Behind the scenes, Rayon uses work stealing to try and dynamically ascertain how much parallelism is available and exploit it. The idea is elementary; there is always a pool of worker threads available, waiting for some work to do. When *join* is called for the first time, we shift over into that pool of threads. However, if $join(a, b)$ is called from a worker thread W, then W will place b into its work queue, advertising that this is work that other worker threads might help out with. W will then start executing a. While W is busy with a, other threads might take b from its queue. That is called stealing b. Once a is done, W checks whether another thread stole b and, if not, executes b itself. If W runs out of jobs in its queue, it will look through the other threads' queues and try to steal work from them. This technique is not new. The Cilk project was first introduced at MIT in the late nineties. The name Rayon is an homage to that work.
In this project the method *join* of Rayon is called with the arguments a, a method to run Glucose and b, a method to parallel build the BDD. As Glucose is written in C, bindings were used to connect the C code to the Rust code and use the C functions. In detail, this project does not use an instance of Glucose; instead, it calls the Glucose functions through functions in the Rust code.
Furthermore, as Glucose and BDD are running in parallel, they need a way to communicate with each other. As the BDD will take longer to build than Glucose to finish, Glucose needs to be able to send a message to the BDD part when it finishes so that the construction of the BDD is terminated as well. Another important message that Glucose needs to pass to the BDD is if it has timed out. For that matter, a crossbeam channel is implemented between the two architectures. The Rust crate crossbeam channel provides multi-producer multi-consumer channels for message passing. It is an alternative to $std :: sync :: mpsc$ with more features and better performance. Some highlights are:

- Senders and Receivers can be cloned and shared among threads.
- Two main kinds of channels are bounded and unbounded.
- Convenient extra channels like after, never, and tick.
- The select! macro can block multiple channel operations.

- Select can select over a dynamically built list of channel operations.
- Channels use locks very sparingly for maximum performance.

The communication between Glucose and BDD is enabled if Glucose holds the Sender and the BDD the Receiver of the crossbeam channel.

5.3 Information Exchange

Not only do messages need to be exchanged between BDD and Glucose but also information, for example, learnt clauses.

In the chapter before, it was stated that the Clause Database 4.3.3 is responsible for the information filtering and exchange for both architectures. Particularly, a clause sent from the BDD is not sent directly to Glucose but instead to the Clause Database. Later, if it passes the Bloom filters 4.3.3 of the Clause Database, it is sent to Glucose. The procedure is equal if a clause is sent from Glucose to the BDD architecture. Bloom filters are used to detect duplicate clauses. A Bloom filter is a space-efficient probabilistic set data structure that allows false-positive matches, meaning that some clauses might be considered duplicates even if they are not.

As the BDD is being constructed, witness clauses are being searched in the current BDD. In such a BDD, each path to the *zero* (false) node denotes a conflict or a witness clause as explained in chapter 4. A learned clause corresponding to this conflict is easily obtained by negating the literals that define the path. Since a BDD captures all paths to *zero*, i.e., all possible conflicts, the potential advantage is that multiple learned clauses can be generated and added to the SAT solver at the same time.
The method *send* presented below is the one sending the learnt clauses from the BDD to Glucose.

```
/// This method sends the clause to the database, where it is filtered
/// and it receives back the clause or an error that the clause
/// did not pass the databases' filters. After that
/// the learned clause has to be sent back to the
/// solvers, but it cannot be sent back to the solver it came from.
pub fn send(&mut self, clause_input: Vec<i32>, solver_wrapper: GlucoseWrapper,
stats: &mut Stats) {
    stats.add_sent_bdd();
    // the clause is registered to the clause database
    if let Ok(learned_clause) = self.get_next_incoming_clause_bloom(clause_input) {
        // the clause passed the filters so send it to Glucose
        let solver = solver_wrapper.0;
        // add the clause to Glucoses' receive_tmp so that glucose catches it from there
        add_incoming_clause_to_clauses_vec(solver, learned_clause);
```

```
            stats.add_received_glucose();
        }
    }
```

The clause is firstly filtered through the Bloom filters in the Clause Database and then sent to Glucose with the help of the bindings between Rust and C. Glucose contains a vector called *add_tmp_receive*. It is a temporary vector, which enables the BDD to write every learnt clause in this vector, and then Glucose processes them from there. In detail, the vector is cleared, the newly learnt clause is written to the vector, and the clause is committed to Glucose. The following methods demonstrate the process described above.

```
pub fn add_incoming_clause_to_clauses_vec(s : *mut CGlucose, given : Vec<i32>){
    unsafe {
        cglucose_clean_clause_receive(s);
        for i in given{
            cglucose_add_to_clause_receive(s, i as i32);
        }
        cglucose_commit_incoming_clause(s);
    }
}

void SimpSolver::commitIncomingClause() {
    if (add_tmp_receive.size() != 0) {
        CRef cr = ca.alloc(add_tmp_receive, true, true);
        ca[cr].setLBD(add_tmp_receive.size());
        ca[cr].setOneWatched(false);
        attachClause(cr);
        add_tmp_receive.clear();
    }
}
```

The process of sending a learnt clause from Glucose back to the BDD solver is similar. A learnt clause is encountered through Glucose's conflict analysis. This clause is then written to a temporary vector called *add_tmp_send*. The C clause gets translated to a Rust clause and is attached to the BDDs clauses to be processed. The translation procedure is shown in the code below.

```
pub fn get_exported_clause_from_glucose(s : *mut CGlucose) -> Option<Vec<i32>> {
    let size = get_exported_clause_size(s);
    if size == 0 {
        None
    } else {
        let mut exported_clause = Vec::new();

        let mut pos = 0;
```

```
        while pos < size {
            let lit = get_exported_lit_at(s, pos);
            exported_clause.push(lit);
            pos += 1;
        }
        unsafe { cglucose_clean_clause_send(s); }
        Some(exported_clause)
    }
}
```

The goal of this information exchange is to not only support the CDCL process by sending learnt clauses from the BDD to Glucose but also to help accelerate the construction by sending learnt clauses from Glucose to the BDD.

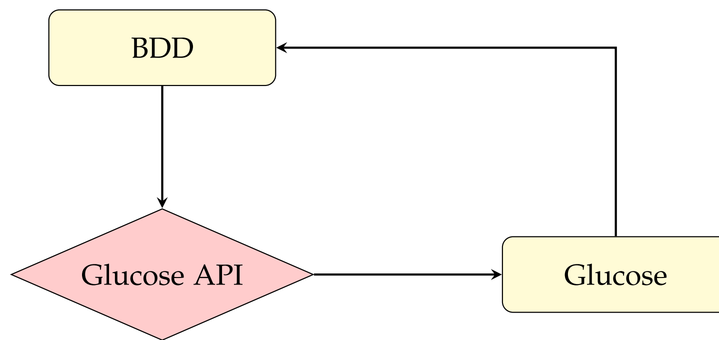


Figure 5.1: Communication between the architectures

6 Performance and Measurements

The main contribution of this thesis is the comparison of Glucose SAT solver with and without the support of the BDD library implemented in Rust, based on a supply of generated benchmarks from the annual SAT competitions.

6.1 Benchmarks

At the annual International Conference on Theory and Applications of Satisfiability Testing, a SAT Competition is held. The benchmarks of this Competition are freely available online and can be used to evaluate the developed solvers. All benchmarks are given as SAT problems in DIMACS format. Detailed descriptions of the benchmarks and solvers are published in the Proceedings of SAT Competition of the respective year after that. The benchmarks can be divided into the following three categories:

Problems from Applications Problems from the industry come from areas such as design and verification. At the 2013 SAT Competition, this category of benchmarks included 2D strip packing, bounded model checking, cryptographic applications, and scheduling problems. These formulas often have a high number of variables and clauses. The 600 instances of this category in the 2012 SAT Competition have an average of 279026 variables and 1735634 clauses. Modern SAT solvers and CDCL specialize in this category as they have applications.

Combinatorial Problems The formulas in this category come from areas of discrete mathematics and number theory, such as calculating Van der Waerden numbers or graph isomorphism problems. It also includes problems with fewer applications whose solution space explodes combinatorially, for example, solving games like Sudoku or sinking ships. It also included being constructed worst-case examples of SAT algorithms. For example, it is known that so-called pigeonhole problems are challenging for CDCL solvers. The combinatorial problems usually manage with much fewer variables. Again at the SAT Competition 2012, one gets an average number of only 10759 variables and 78997 clauses.

Random formulas Formulas that do not come from any concrete problem but were generated uniformly at random, for example. Random formulas have interesting properties; for example, the satisfiability threshold conjecture states that a random formula alone is (not) satisfiable with a high probability depending on the ratio of clauses to variables. Such properties allow solvers to be specially adapted for this category, which is not the case with our developed solvers.

6.1.1 Test Cases

Since SAT solvers have made significant progress in recent years, the benchmarks have naturally become more complex. At the SAT Competition 2019, there is a time limit of 5000 seconds (over 1h 20min) per input, with up to 128 GB RAM.
This project utilized benchmarks from the SAT Competitions from 2008 until 2013. The benchmarks from 2008 until 2012 have a time limit of 900 seconds, whereas the ones from 2013 have a time limit of 5000 seconds. The benchmarks tested in this project were the following:

- over 300 random benchmarks from 2008 and 2009
- 400 random benchmarks from 2012
- 22 hard/combinatorial benchmarks from 2012
- 10 hard/combinatorial benchmarks from 2013
- 7 application benchmarks from 2012

As the test cases have a time limit of either 900 or 5000 CPU seconds, it is clear that not all of them terminated. In section 6.3 the results from the hard/combinatorial benchmarks from 2012 will be presented and analyzed, as they are the more indicative ones.

6.2 Testing Environment

The tests were conducted remotely on The Computer Operations Group (RBG) of the Institute of Computer Sciences of the LMU Munich through ssh services. The operating system of the workstation computers is Ubuntu 18.04 LTS, and the equipment is the following:

- act. CPU with 4 cores + HT
- 64GB Ram
- 256GB NVMe SSD
- fast network connection
- Nvidia 1050TI or 1060
- 4k monitor

6.3 Runtime and Measurements

The aim of this Master's thesis is to compare Glucose running in parallel with the BDDs and alone and show that the BDD library implemented is supporting the CDCL process and improving Glucose's performance and the real-time communication between the two

architectures is efficient. For the conduction of the tests, Glucose with the support of the BDDs and Glucose alone were running in parallel, with the help of the Rust crate Rayon, which was introduced in the section 5.2. It is crucial to execute the two procedures in parallel and in the same testing environment to get accurate results.
In order to get a concrete analysis of both Glucose with and without the support of the BDD solver, the runtimes of every operation (CPU and world) were measured. Additionally, the number of restarts 3.4.1, propagations 3.2, conflicts, and decisions 3.3 in both cases were counted. The figures in section 6.5 represent the final results of running the benchmarks, using the CDCL SAT solver with and without the BDD library implemented. The analysis and discussion of these results can be found in section 6.5, whereas the conclusion of this work can be located in chapter 7.
It is essential to mention that several testing parameters were set and adjusted to reach the best results. For instance, the BDD was approximated firstly every time after processing two clauses, then every ten clauses, and lastly every thirty clauses. The conclusion was drawn that the most suitable configuration is to approximate the BDD every ten clauses. Moreover, the global Bloom filter 4.3.3 is reset after processing 30 clauses. After several experiments, it was discovered that it is most reasonable to reset the global filter and transmit some clauses again to Glucose, as in the later progress of Glucose, they might be helpful.

6.4 Code and Data

A large part of the work behind this thesis involved writing the source code for the various algorithms described. The implementation consists of the BDD library written in Rust, the communication and information exchange between the two architectures (Glucose and BDD), as well as the bindings between the C code of Glucose and the Rust interface. The code is available for download in Gitlab under BDD Sat Solver.

6.5 Analysis

This section presents the results from running the hard/combinatorial benchmarks simultaneously on both Glucose with the support of the BDDs and Glucose alone in the same testing environment. After a summary is provided, the results are also explained and discussed. Cactus plots have been produced to show the general results. A cactus plot or "survival plot" is used to summarize the performance of an automated verification tool in verification tool competitions. It is important to note that the statistics consist of runtime (CPU), number of conflicts, restarts 3.4.1, propagations 3.2, and decisions 3.3. From the hard/combinatorial benchmarks from 2012 tested, 22 are presented in the graphs. The configuration used while conducting these tests was approximation every ten clauses and resetting the global Bloom filter 4.3.3 every thirty clauses.

Figure 6.1 shows the results of the CPU runtime from the 22 instances. The red line represents Glucose alone, and the green line Glucose with the support of the BDDs. As mentioned

before, the timeout was set at 900 seconds. Looking at the cactus plot, we can see that overall, the CPU runtime when running Glucose and BDDs in parallel is lesser than the CPU runtime when running Glucose alone. We can also observe that when running Glucose alone, five instances are over the timeout limit, as instances, number 18 and 19 are over 900 seconds, and instances number 20, 21, and 22 are not shown in the chart as they did not terminate. When running Glucose and BDD in parallel, 21 instances terminated before the timeout, and just one terminated at 973 seconds, slightly above the time limit.

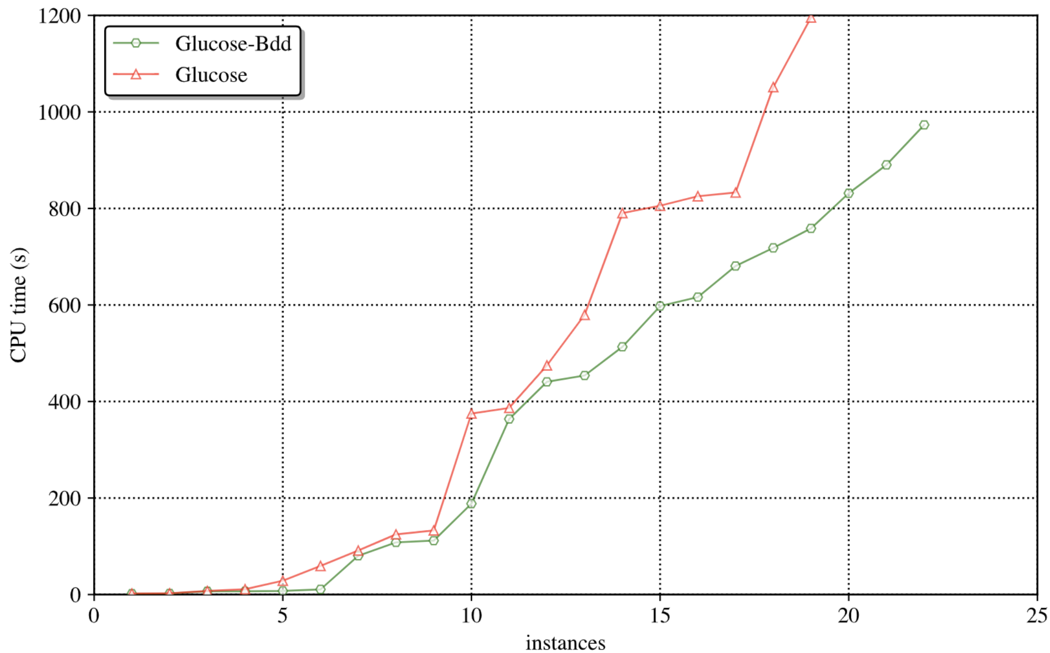


Figure 6.1: CPU Time

The figures pictured below show the number of conflicts, restarts, decisions or propagations from running 22 instances both on Glucose alone and Glucose in parallel with the BDDs. The red line represents Glucose alone, and the green line Glucose with the support of the BDDs. As mentioned before, from the instances represented, 17 terminated on time when running Glucose alone, from which 19 are pictured as the last two terminated a few seconds after the timeout, and the other three did not terminate. When running Glucose and BDDs in parallel, 21 instances terminated within the time limit and one slightly after; that is why the results from 22 instances are shown.

Figure 6.2 represents the number of conflicts from running the 22 instances both on Glucose alone and with the support of the BDDs. Looking at the cactus plot, we can see that the green line is below the red line, which means that when running Glucose with the BDDs, fewer conflicts happened in the exact instances. It is also worth mentioning that as the instances become larger, which is the case after instance number 12 in the plot, the difference in the

number of conflicts between Glucose and Glucose with BDDs becomes bigger.

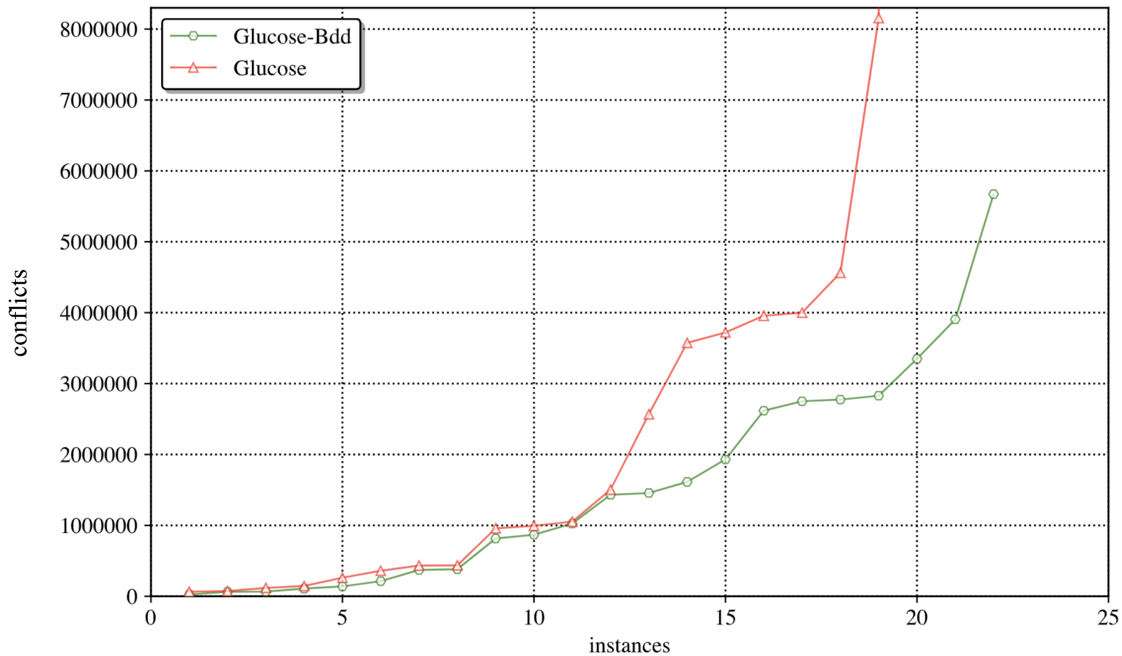


Figure 6.2: Conflicts

Some benchmarks were analyzed in detail while running, and the number of conflicts after several steps was collected each time. The thorough analysis of the benchmark $sgen4-unsat-89-1.cnf$, one of the hard/combinatorial benchmarks from 2012 can be witnessed in the graph 6.3. An essential remark here, which is very important for future work, is that when the BDD size is kept small, the number of conflicts at the CDCL side is also kept low. The number then rises when the BDD size rises. The explanation for this is that the clauses sent to Glucose when the BDD is growing exponentially are not anymore as strong as the small clauses sent in the beginning when the size of the BDD was small. Correspondingly, the bigger the BDD, the longer it takes for operations such as approximation or searching for witness clauses to complete. In future work, the learnt clauses could be simplified before being sent to Glucose.

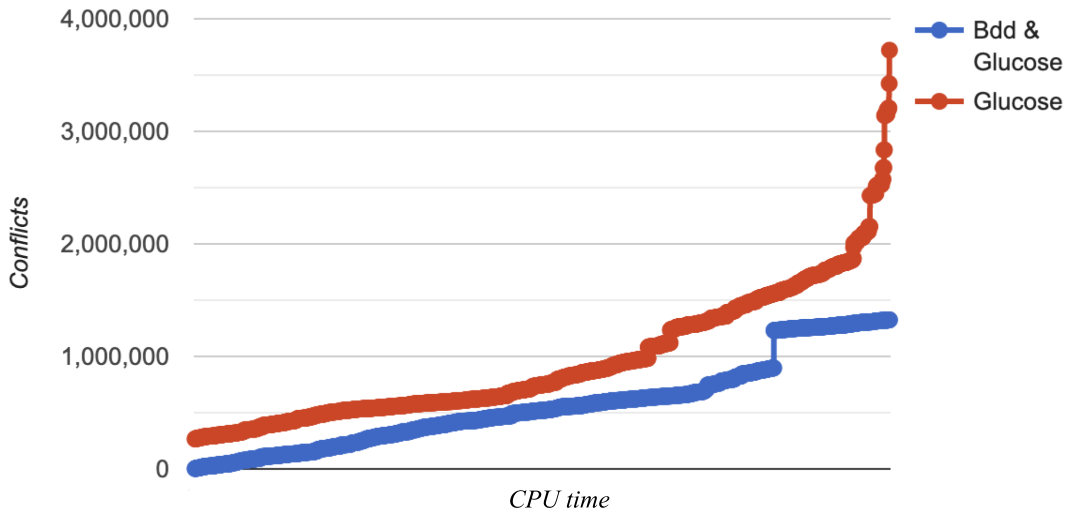


Figure 6.3: Conflicts of sgen4-unsat-89-1.cnf in time

Figure 6.4 portrays the number of restarts from running the 22 instances both on Glucose alone and Glucose in parallel with the BDDs. Looking at the cactus plot, we can see that, generally, the green line is below the red line, so when running Glucose in parallel with the BDDs, rarer restarts happened on the exact instances. Similarly, as the instances become larger or more complicated, the difference in the number of restarts between Glucose and Glucose with BDDs becomes greater as well. It can be seen in the chart that whereas in the beginning, the number of restarts did not differ much, after instance number 14, the two lines grow significantly separate from each other.

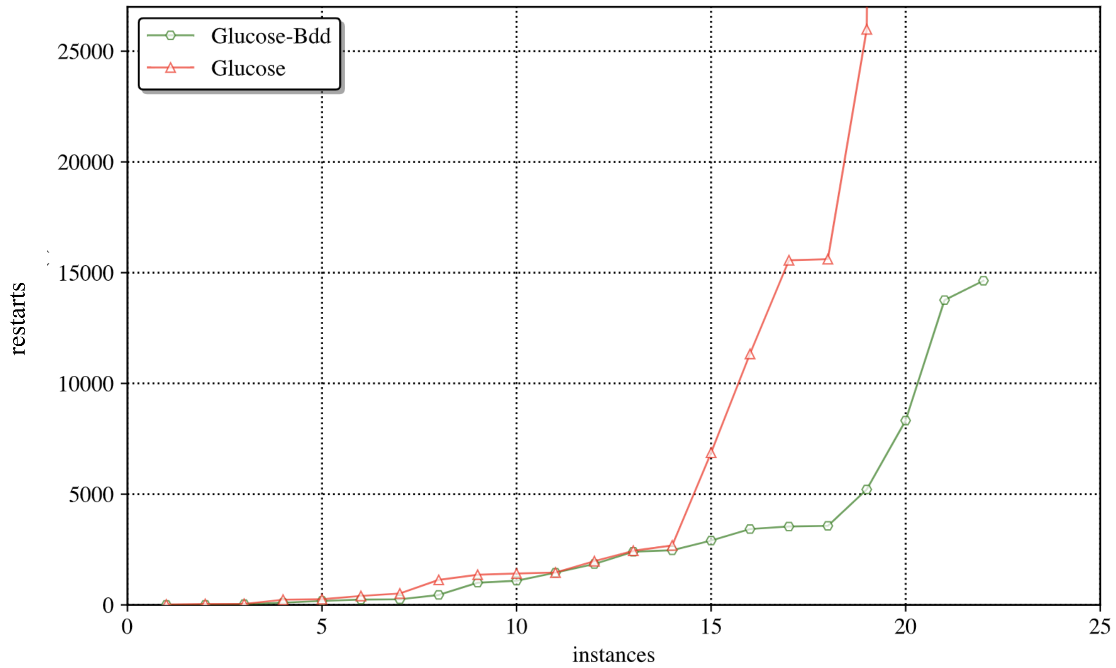


Figure 6.4: Restarts

In figures 6.5 and 6.6 the same behavior as in the graphs above can be observed. When running Glucose in parallel with the BDDs, the final number of decisions and propagations is significantly smaller than when running Glucose alone, especially when the instances are big or complicated. As the hard/combinatorial instances are introduced on the charts, it is often the case that the solvers were tested on challenging conditions. When referring to propagations, the iterated application of the unit clause rule is implied (unit propagation); if a clause is unit, then its sole unassigned literal must be assigned value 1 for the clause to be satisfied. Likewise, the assignment set for a variable at a decision level is suggested when referring to decisions.

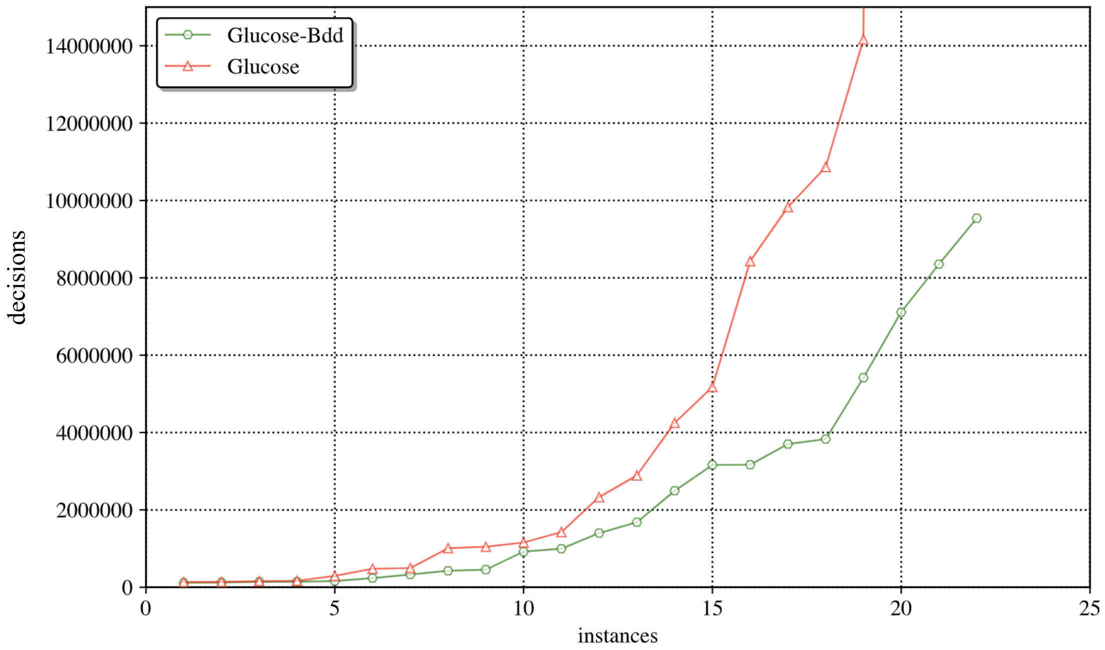


Figure 6.5: Decisions

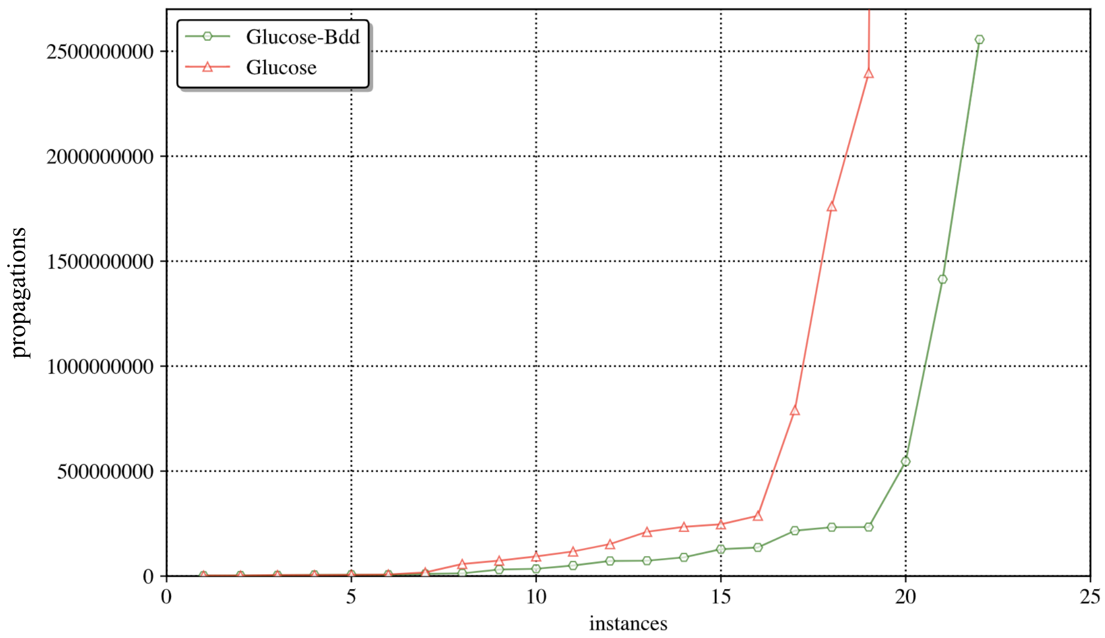


Figure 6.6: Propagations

A necessary observation here is that the BDD, as it is not reaching the end of the clauses vector, does not process the clauses sent from Glucose. If it did, this could help enhance the

BDD performance, and then the BDD on its side would be able to support Glucose even more.

Summarizing, the benchmarks tested provide reliable results as they belong to the hard/-combinatorial category of the 2012 benchmark set. In every chart, Glucose's performance with the support of the BDD solver is better than the performance of Glucose alone. Also, the larger the instances, the better the results the two solvers running in parallel achieve. This can be seen on the graphs, where after instance number 10, the gap between the green and the red line becomes more prominent. Finally, the most promising result is that some of the 2012 and 2013 hard/combinatorial benchmarks timed out when running on Glucose alone but not when running on Glucose with the support of the BDD solver. Hopefully, with the improvements planned for future work and suggested in chapter 8, the number of instances solved with the help of the BDDs can be increased.

7 Conclusion

From the examples mentioned above and after running several benchmarks on Glucose with the support of the BDDs and Glucose alone following conclusions can be drawn:
With the help of the BDDs, the performance of the CDCL SAT solver improves. In detail, the number of conflicts, restarts 3.4.1, decisions 3.3, and propagations 3.2 in the CDCL process drops significantly. Consequently, the CDCL SAT solver has significantly lower computational costs as many redundant conflicts, restarts, decisions, and propagations are avoided. Most importantly, the CPU time needed for the CDCL solver to terminate is remarkably lower with the support of the BDD solver, which results in more instances terminating before the set time limit. Ally, the CDCL SAT solver becomes more efficient.
Additionally, the core BDD operations, such as searching for witness clauses and sending the learnt clauses to Glucose and approximation, cost time. However, even with the approximation and searching of learnt clauses operation being not that efficient, the overall performance of the BDD supporting Glucose is exemplary, and the results in computation time are favorable. It is safe to say that if the core algorithms were improved in future work, the overall performance would too.
Finally, from the latest hard/combinatorial benchmarks of 2012 and 2013, **4/22** from 2012 and **2/10** from 2013 did terminate before timeout when the two architectures were running in parallel and timed out when not. In a first examination, this occurred due to the amount of small strong clauses sent from BDD to Glucose until the BDD started rising exponentially in size. Few benchmarks were analyzed in detail while running, and the number of conflicts after several steps was collected each time. As the size of the BDD in the beginning is small, the learnt clauses encountered and sent to Glucose are also small. Smaller clauses are more potent than the bigger ones, so the CDCL process encountered fewer conflicts when processing the smaller learnt clauses. As the BDD size rises exponentially, the witness clauses become bigger, and the number of conflicts also rises. This will be, of course, further investigated. Further, while the BDD sends many clauses, a small number passes the Bloom filter 4.3.3 and are sent to Glucose. Precisely, in future improvements, it will be beneficial to simplify the learnt clauses before sending them to the CDCL solver and also adjust the Bloom filters to let the stronger clauses pass.

8 Future Work and Suggestions

This thesis promoted the overall work until a certain point leading the way for further investigation. In this chapter, the future work and suggestions planned for after submitting this Master's thesis will be addressed. Ideas for improvement will be expressed, which, when implemented, will enhance the project's performance. In chapters 6 and 7 few ideas for improvement have been presented, derived from the results of running the benchmarks on both Glucose alone and Glucose with the support of the BDD solver.
First of all, the communication between the two architectures is achieved through shared vectors. In detail, both BDD and Glucose send a learnt clause to the Clause Database after finding it. If the clause passes the Bloom filters, it gets written on a shared vector and then directly on the vector of clauses to be processed by every architecture respectively. A crossbeam channel is implemented to notify the BDD if Glucose has terminated. The idea for future work is to implement crossbeam channels between BDD and Clause Database and Glucose and Clause Database. Each architecture will have a Sender and a Receiver from the channel. In that way, the Sender will send the learnt clauses to the Clause Database, while at the same time, the Receiver will receive the learnt clauses sent from the other architecture from the Clause Database. The Message Parsing Interface (MPI)s will also include the Bloom filters. Communication will become more efficient in this manner. The implementation for a communication like this was initiated in this project but needed to be moved to the future plans as the bindings between C and Rust complicate it.
Moreover, the core operations of the BDD need to become more efficient. As noticed while running the benchmarks, operations like the approximation and the search for witness clauses cost time and need to be optimized. Particularly, the results have shown that when the BDD is small, the clauses sent over to Glucose are smaller and, therefore, more powerful. Suppose the BDD does not increase exponentially in size; the operations mentioned above, for instance, the approximation and the search for witness clauses, will be more efficient as the search tree will be smaller. Also, more approximation techniques can be implemented, accelerating the approximation operation and reducing the Binary Decision Diagram (BDD) size even more. For that reason, future plans include implementing more approximation techniques and a method to simplify the learnt clauses before sending them to Glucose so that it is ensured that the clauses are useful for CDCL.
Additionally, while the BDD sends many clauses, a small number passes the Bloom filter 4.3.3 and are sent to Glucose. In future improvements, it will be beneficial to not generate so many learnt clauses at the BDD and focus more on generating smaller, stronger clauses to send to the CDCL solver. Furthermore, it is also important to adjust the Bloom filters to let the stronger clauses pass, or even implement a new Bloom filter with the needed specifications. Lastly, as the BDD does not terminate, it does not reach the end of the vector of clauses to be

processed. This suggests that the learnt clauses sent from Glucose and attached to the end of that vector never enter the BDDs algorithms. Therefore, a method needs to be encountered which will process the learnt clauses sent from Glucose directly so that the CDCL procedure can support the BDD as well.

List of Figures

2.1 Truth Tables 3
2.2 A Decision Tree for $(x_1 \iff y_1) \wedge (x_2 \iff y_2)$ 6
2.3 An Ordered Binary Decision Diagram with Variable Ordering $x_1 < y_1 < x_2 < y_2$ 7
2.4 An Ordered Binary Decision Diagram with Variable Ordering $x_1 < x_2 < y_1 < y_2$ 7

3.1 Implication Graph 15
3.2 Implication Graph 16
3.3 Implication Graph 16

4.1 Bdd Library Architecture 20
4.2 An Reduced Ordered Binary Decision Diagram with Variable Ordering $x_1 < y_1 < x_2 < y_2$ 26
4.3 An Reduced Ordered Binary Decision Diagram from clauses $\{x_1 \vee x_4\}$ and $\{x_2 \vee x_{11}\}$. 28
4.4 An Reduced Ordered Binary Decision Diagram before rounding. 30
4.5 An Reduced Ordered Binary Decision Diagram after rounding. 30
4.6 An Reduced Ordered Binary Decision Diagram from clauses $\{x_1 \vee x_4\}$ and $\{x_2 \vee x_{11}\}$. 31

5.1 Communication between the architectures 36

6.1 CPU Time 40
6.2 Conflicts 41
6.3 Conflicts of sgen4-unsat-89-1.cnf in time 42
6.4 Restarts 43
6.5 Decisions 44
6.6 Propagations 44

Glossary

BDD Binary Decision Diagram. 36, 49

CDCL Conflict Driven Clause Learning. 49

CNF Conjunctive Normal Form. 49

DAG Directed Acyclic Graph. 49

DNF Disjunctive Normal Form. 49

DPLL Davis–Putnam–Logemann–Loveland. 49

MPI Message Parsing Interface. 49

OBDD Ordered Binary Decision Diagram. 49

ROBDD Reduced Ordered Binary Decision Diagram. 49

SAT Boolean Satisfiability. 49

Acronyms

BDD Binary Decision Diagram. iv, v, 1–6, 18–49

CDCL Conflict Driven Clause Learning. iv, v, 1, 2, 11–14, 17, 22, 27–30, 32, 36–39, 41, 46–49

CNF Conjunctive Normal Form. 2, 4, 11, 12, 19, 25–27, 30, 31, 49

DAG Directed Acyclic Graph. 6, 49

DNF Disjunctive Normal Form. 4, 49

DPLL Davis–Putnam–Logemann–Loveland. 11–13, 17, 49

MPI Message Parsing Interface. 19, 20, 47, 49

OBDD Ordered Binary Decision Diagram. 6–8, 23, 24, 29, 49

ROBDD Reduced Ordered Binary Decision Diagram. 3, 7, 8, 23, 28, 29, 49

SAT Boolean Satisfiability. iv, v, 1, 2, 11–13, 17, 19, 20, 22, 25, 28, 29, 32, 34, 37–39, 46, 49

Bibliography

[1] B. Kell, A. Sabharwal, and W. J. van Hoeve. “BDD-Guided Clause Generation”. In: *Integration of AI and OR Techniques in Constraint Programming*. 2015.

[2] T. Balyo, P. Sanders, and C. Sinz. *HordeSat: A Massively Parallel Portfolio SAT Solver*. 2015. DOI: 10.48550/ARXIV.1505.03340. URL: https://arxiv.org/abs/1505.03340.

[3] Y. Hamadi, S. Jabbour, and L. Sais. “ManySAT: a Parallel SAT Solver”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 6.4 (June 2009). Ed. by Y. Hamadi, pp. 245–262. DOI: 10.3233/sat190070. URL: https://doi.org/10.3233%5C%2Fsat190070.

[4] F. Leimgruber. *BDD-unterstütztes SAT-Solving*. May 2020.

[5] R. E. Bryant. “Symbolic Boolean manipulation with ordered binary-decision diagrams”. In: *ACM Computing Surveys* 24.3 (Sept. 1992), pp. 293–318. DOI: 10.1145/136035.136043. URL: https://doi.org/10.1145/136035.136043.

[6] H. R. Andersen. “An Introduction to Binary Decision Diagrams”. In: 1997.

[7] H. Moeinzadeh, M. Mohammadi, H. Pazhoumand-dar, A. Mehrbakhsh, N. Kheibar, and N. Mozayani. “Evolutionary-Reduced Ordered Binary Decision Diagram”. In: *2009 Third Asia International Conference on Modelling Simulation*. IEEE, 2009. DOI: 10.1109/ams.2009.130. URL: https://doi.org/10.1109/ams.2009.130.

[8] Mozilla. *Rust language*. 2016. URL: https://research.mozilla.org/rust/.

[9] A. Biere, M. Heule, H. van Maaren, and T. Walsh, eds. *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009. ISBN: 978-1-58603-929-5.

[10] J. Marques-Silva, I. Lynce, and S. Malik. “Conflict-Driven Clause Learning SAT Solvers”. In: *Handbook of Satisfiability*. 2021.

[11] C. Gomes, B. Selman, N. Crato, and H. Kautz. “Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems”. In: *Journal of Automated Reasoning* 24 (Jan. 2000). DOI: 10.1023/A:1006314320276.

[12] B. H. Bloom. “Space/time trade-offs in hash coding with allowable errors”. In: *Communications of the ACM* 13.7 (July 1970), pp. 422–426. DOI: 10.1145/362686.362692. URL: https://doi.org/10.1145/362686.362692.

[13] K. Pipatsrisawat and A. Darwiche. “On the power of clause-learning SAT solvers as resolution engines”. In: *Artificial Intelligence* 175.2 (2011), pp. 512–525. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2010.10.002. URL: https://www.sciencedirect.com/science/article/pii/S0004370210001669.

[14] H R Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. “A Constraint Store Based on Multivalued Decision Diagrams”. In: (2018). DOI: 10.1184/R1/6702881.V1. URL: https://figshare.com/articles/A_Constraint_Store_Based_on_Multivalued_Decision_Diagrams/6702881/1.

[15] S. Froehlich, D. Große, and R. Drechsler. “Error Bounded Exact BDD Minimization in Approximate Computing”. In: *2017 IEEE 47th International Symposium on Multiple-Valued Logic (ISMVL)*. 2017, pp. 254–259. DOI: 10.1109/ISMVL.2017.11.

[16] S. Gopalakrishnan, V. Durairaj, and P. Kalla. “Integrating CNF and BDD based SAT solvers”. In: *Eighth IEEE International High-Level Design Validation and Test Workshop*. 2003, pp. 51–56. DOI: 10.1109/HLDVT.2003.1252474.

[17] G. Audemard and L. Simon. “Predicting Learnt Clauses Quality in Modern SAT Solvers”. In: *Proceedings of the 21st International Joint Conference on Artificial Intelligence*. IJCAI’09. Pasadena, California, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 399–404.