Development of a Parallelized BDD Library in Rust

Timo Netzer
Timo Netzer:  
*Development of a Parallelized BDD Library in Rust*

Advisors:  
- M.Sc. Tobias Heß  
  *Institute of Software Engineering and Programming Languages*
- Prof. Dr. Thomas Thüm  
  *Institute of Software Engineering and Programming Languages*


This thesis is available at the Open Access Repository of Ulm University and Ulm University of Applied Sciences (https://oparu.uni-ulm.de) and published under the CC BY 4.0 license. Further informations about the license can be found at https://creativecommons.org/licenses/by/4.0/.
Abstract

Binary decision diagrams (BDD) are data structures that represent Boolean functions. Two of the most fundamental properties of BDDs are the abilities to solve the satisfiability problem (SAT) and count the number of solutions for the satisfiability problem ($\#\text{SAT}$) efficiently. Additionally, the Boolean function NOT can be applied in constant time and the Boolean functions AND and OR can be applied in polynomial time. While SAT can be solved in constant time, $\#\text{SAT}$ can be solved in linear time, with respect to the number of nodes in the BDD. Solving SAT and $\#\text{SAT}$ efficiently is important for many real world applications, such as product-line analysis and circuit analysis. Currently, there exist several libraries to generate BDDs from Boolean functions, the most widespread being BuDDy and CUDD, both whom 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 powerful than BuDDy and CUDD. In this thesis, we address the shortcomings by providing an architecture for a BDD library with not only parallelization in mind, but additionally, we want the architecture to be modular so the different algorithms used for the construction of the BDD can be exchanged to measure runtime impacts of different implementations. We also give high priority to maintainability and ease of use. As a result of the design process, we provide an architecture that meets the specified points of parallelism, extensibility, maintainability and ease of use, ensured by software design patterns, like the facade pattern and the pipes-and-filters pattern. Furthermore, we provide a proof-of-concept implementation as a minimal viable product of the described architecture in the Rust programming language which can construct BDDs using parallelism, solve SAT and $\#\text{SAT}$ and provide manipulation through Boolean functions.
Contents

List of Figures vii
List of Tables ix
List of Code Listings xi

1 Introduction 1

2 Background 3
  2.1 Boolean Functions ........................................ 3
  2.1.1 Shannon Decomposition ................................ 4
  2.1.2 Conjunctive Normal Form and Disjunctive Normal Form .... 4
  2.2 SAT and #SAT ............................................ 5
  2.3 Binary Decision Diagrams ................................ 5
  2.3.1 Ordered Binary Decision Diagrams ...................... 6
  2.3.2 Reduced Ordered Binary Decision Diagrams .......... 7
  2.4 Data on the Heap and Stack ................................ 7

3 Library Design 11
  3.1 Applied Software Design Patterns ........................ 12
  3.2 Modular Components ...................................... 13
  3.3 Summary .................................................. 16

4 Implementation of the Library 17
  4.1 Implementation Basics .................................... 17
  4.1.1 Hash Tables ........................................... 17
  4.1.2 General Boolean Function API by Bryant ............ 18
  4.1.3 If-Then-Else ......................................... 19
  4.2 Rust Implementation Details ............................... 22
  4.2.1 Mutability ............................................. 22
  4.2.2 Ownership and Borrowing ............................. 23
  4.2.3 LLVM Backend ......................................... 25
  4.2.4 Parallelization ....................................... 25
  4.2.5 Rust Standard Hash Map ............................... 26
  4.2.6 (Atomic) Reference Counted Smart Pointer ....... 26
List of Figures

2.1 A truth table and a binary decision tree representing the NAND function. 4
2.2 A BDD with annotations representing the NAND (¬x₁ \lor ¬x₂) function. 6
2.3 Two BDDs representing the same Boolean function, showing the difference between variable orders [Bey05]. 7
2.4 A diagram showing the application of reduction rules (R₁) and (R₂) on a BDD. 8
3.1 A diagram for the pipeline-like structure of the library. 12
4.1 An overview of the anatomy of a hash table. 18
4.2 A figure showing the application of AND to two BDDs. 22
4.3 A figure showing the application of NOT to a BDD. 22
4.4 The graph for measurements for sequential BDD execution in the prototype. 36
List of Tables

4.1 A table containing the average and worst case complexities of hash tables. ............................................................. 18
4.2 The adjusted boolean function API as proposed by Bryant. ............. 19
4.3 Every two-variable Boolean function written in terms of the ITE function. ............................................................. 21
List of Code Listings

2.1 A code example for failing unlimited size data structure. .................. 9
2.2 A code example for the usage of heap allocated memory in a ListElement. .............................. 9
3.1 The example DIMACS CNF file provided by DIMACS. ..................... 15
4.1 A pseudo code implementation for the ITE function. ...................... 20
4.2 A code example for Rust immutability. ........................................ 23
4.3 A code example for Rust mutability. .......................................... 23
4.4 A code example for Rust ownership. ........................................... 23
4.5 A code example for movement of ownership in Rust. ...................... 24
4.6 A code example for a function that takes ownership of a variable. ... 24
4.7 A code example for a function that borrows a value. ..................... 24
4.8 Examples for function signatures for each of the different ownership states in Rust. .................................................. 25
4.9 A code example for the usage of Rc. ........................................... 27
4.10 A code example for the usage of Arc. ....................................... 27
4.11 An implementation of the factorial function with test cases in Rust. 29
4.12 An overview of important definitions in parser.rs ...................... 31
4.13 An overview of important definitions in boolean_function.rs ....... 31
4.14 An overview of important definitions in static_ordering.rs ......... 32
4.15 An overview of important definitions in bdd_manager.rs ............ 33
4.16 An overview of the Manager trait. .......................................... 34
4.17 An excerpt of the Rust documentation for the Manager trait. ....... 38
4.18 A serialized BDD in our custom format. ................................. 39
4.19 The standard output after an invocation of the OBDDimal CLI tool without parameters. ......................... 39
1. Introduction

A binary decision diagram (BDD) is a data structure that is used to represent Boolean functions. Formally a BDD is a rooted, directed, acyclic graph [Knu09]. A BDD consists of nodes that represent variables in a Boolean function and two terminal nodes that represent the values true and false. Every Boolean function can be represented by a BDD [Knu09].

BDDs can be used to solve the satisfiability problem (SAT) for a given Boolean function. SAT is the problem of determining if there exists a set of inputs for a Boolean function which evaluates the function to true. Counting the number of satisfying variable assignments of different input sets evaluating the function to true is called the sharp satisfiability problem (#SAT) [Val79].

A BDD can solve SAT in constant time and #SAT in linear time (with respect to the number of nodes of the BDD) [Knu09]. This improvement of time complexity is possible by shifting the time complexity from the evaluation of SAT and #SAT to the construction of the data structure [DMo2], this is known as knowledge compilation [DM02, Tho].

Many real-world problems can be approached by expressing them as a Boolean function and solving the SAT for them, for example, product-line analysis [BSRC10], cryptanalysis [SN19], dependency resolution of package managing systems [Sch21], term-rewrite systems [MS08], circuit-level prediction of crosstalk noise [MS08], and Sudoku solvers [Web]. Moreover, there are real-world problems that can be solved using #SAT, for example calculating the variability factor of a feature model [BRCT05, SNB+21], predict the maintainability of a feature model [BNGR12, SNB+21] and calculate the cost savings of a product line [CMCO5, HFACA13, SNB+21]. However, it is well known that SAT is NP-complete [ST13] and #SAT is #P-complete [ST13], which means there is no known algorithm that solves SAT or #SAT in polynomial time [ST13, Coo71], therefore finding solutions with naive approaches are not feasible, even for smaller input sizes [GJ79, Laa18].
When shifting the time complexity to the construction process of the BDD it is of the highest importance to do the construction as efficient as possible. There exist many implementations of BDD libraries, two of the most well-known ones are BuDDy [BuD] and CUDD [CUD], both are rated as good libraries by Geert Janssen in their comparison of thirteen different BDD libraries [Jan03], besides the analysis by Geert Janssen, both implementations are state-of-the-art. [FABAEH20, FAHMDE19, JS19, JL10, LCF\textsuperscript{+}12, LSX13, PLP11] Because both, BuDDy and CUDD were created with the C or C++ programming languages, which were designed in a time when parallelism in programming was not considered mainstream [S:P12], they construct the BDD sequentially, hence ignoring a potential performance gain by parallelism [SB96].

In this thesis, we describe an architecture for a modular, extensible, maintainable, easy to use, and parallelized BDD library. Modularity and extensibility are granted by a pipeline-like design where every stage of the pipeline is a module, so we can exchange the input format and the variable ordering algorithms. Maintainability and ease of use is ensured by the use of software design patterns [GHJV95, FW20]. We also provide a proof of concept implementation of the described architecture to ensure the architecture is applicable to a software prototype. To reduce the possibility of memory bugs resulting from parallel code execution we use the Rust programming language which is designed for parallelization and with memory safety in mind [KN18].

**Goal of this Thesis**

The goal of this thesis is the conception of a modular and extensible architecture of a parallelized BDD library. We implemented a proof of concept prototype for the proposed architecture in the Rust programming language.

**Structure of the Thesis**

In Chapter 2 on the facing page, we provide the background information needed to understand the concepts of this thesis. In Chapter 3 on Page 11, we discuss the architecture of the library and examine the working method of each part of the library. In Chapter 4 on Page 17, we give an overview of the actual implementation of the prototype, which programming techniques are used and their implementation using the Rust programming language. In addition, we discuss which problems were encountered during the development process and which of those problems remain in the final prototype. In Chapter 5 on Page 41, we compare related work in the sense of already existing BDD packages in terms of similarity with each other, and what the differences between those packages and the subject of this work are. In Chapter 7 on Page 47, we examine still remaining problems and discuss whether they can be solved in future works. Additionally, we outline possible solutions to the persisting problems. Finally, in Chapter 6 on Page 45, the thesis is wrapped up with a conclusion.
2. Background

In the present section, we discuss the fundamental topics necessary to understand a Parallelized BDD Library in Rust and its functions and properties. First, we introduce the concept of Boolean functions and define SAT and $\#\text{SAT}$. Then, we examine binary decision diagrams and different ways a computer can manage memory.

2.1 Boolean Functions

A Boolean function $f : \{0, 1\}^n \to \{0, 1\}, n \in \mathbb{N}$ is a function taking $n$ inputs and mapping them to a single Boolean value 0 or 1, also known as false and true respectively. Multiple ways exist to represent a Boolean function. Here, we concentrate on two of them, the truth table and the binary decision tree. Both of those forms are equivalent, thus we can transform one form into the other.

For example, consider a common Boolean function, the not-and function (NAND).

$$\text{NAND}(x_1, x_2) = \neg(x_1 \land x_2) = \neg x_1 \lor \neg x_2$$

Now, the NAND function can be represented by a truth table or a binary decision tree respectively by evaluating every possible input. As shown in Figure 2.1, both representations are exhaustive, meaning every possible input is explicitly listed. The truth table is self-explanatory. On every row, the values for $x_1, x_2$ and the NAND output are listed. In order to create a binary decision tree, we start with an arbitrary input variable as the root node for the binary decision tree, in Figure 2.1 we began with $x_1$. Next, we determine the zero-edge and the one-edge and connect them to the nodes labelled with the next variable of the function recursively until all input variables are used. The last step is the connection of the last children of the tree and the terminal nodes via a zero-edge and an one-edge. After the construction
of the tree, we add values to the terminal nodes by following the path we receive when evaluating a node that represents a variable either with 0 or 1.

If a truth table and a binary decision tree obey the same variable order the result column of the truth table read from top to bottom is identical to the terminal nodes from the binary decision tree read from left to right. Since every possible input is encoded in those representations, the size of the tree and truth table grows in $O(2^n)$ where $n$ is the number of input variables [Knu09].

![Figure 2.1: A truth table and a binary decision tree representing the NAND function.](image)

### 2.1.1 Shannon Decomposition

The Shannon decomposition states that the following identity holds for every Boolean function $F$ [Boo].

$$F = (x \land F_x) \lor (\bar{x} \land F_{\bar{x}})$$

where $F$ is any Boolean function, $x$ is a variable, $\bar{x}$ is the complement of $x$ and $F_x$ and $F_{\bar{x}}$ is the function $F$ evaluated for $x = 1$ and $x = 0$ respectively. This theorem serves as the foundation for the BDD data structure.

### 2.1.2 Conjunctive Normal Form and Disjunctive Normal Form

Every Boolean function can be written in its conjunctive normal form (CNF) [ST13], where it is expressed as an AND of ORs. Likewise, every Boolean function has a disjunctive normal form (DNF), this is the case when it is written as an OR of ANDs. Formally, we can define those two forms like this:

**Definition 2.1.** Let $t$ be a Boolean term in the form $(x_0 \lor \cdots \lor x_j)$. A Boolean function $BF$ is in its conjunctive normal form if it is written in the following form:

$$BF = \bigwedge_{i=0}^{j} t_i.$$
Definition 2.2. Let \( t \) be a Boolean term in the form \( (x_0 \land \cdots \land x_j) \). A Boolean function \( BF \) is in its disjunctive normal form if it is written in the following form:

\[
BF = \bigvee_{i=0}^{j} t_i.
\]

2.2 SAT and \#SAT

To examine if a Boolean function is satisfiable, we take a look into SAT and \#SAT. SAT stands for Boolean satisfiability problem and it is the problem of deciding whether there exists a set of inputs for a given Boolean function evaluating the function to 1. Then, we say those inputs satisfy the function. Formally, we define a satisfiable function as follows:

Definition 2.3. A function \( f \) is satisfiable if there exists a set of inputs \( (x_1 \ldots x_n), x \in \{0, 1\} \) such that:

\[
f(x_1 \ldots x_n) = 1.
\]

If a function is not satisfiable it is called unsatisfiable.

\#SAT (called Sharp-SAT) is the problem of counting the number of inputs for a given function that satisfy it. Formally, we define \#SAT as:

Definition 2.4. For a Boolean function \( f(x_1 \ldots x_n) \), let \( S \) be the set of all inputs \( x_1 \ldots x_n, x \in \{0, 1\} \) that satisfy \( f \)

\[
\#SAT(f) = x \in \{0, 1\}^n | f(x) = 1.
\]

A function with \( \#SAT(f) = 0 \) is unsatisfiable, and vice versa.

We emphasize that, once we have solved \#SAT for a given function, we have also solved SAT for it (if \( \#SAT > 0 \) the function is satisfiable) [ST13]. However, no known efficient algorithm to solve these problems for a given Boolean function exists at this point [BHIR95].

2.3 Binary Decision Diagrams

A binary decision diagram (BDD) is an additional data structure that represents a Boolean function. A BDD is defined as a rooted, directed, acyclic graph [Knu09]. Such a graph contains decision nodes and terminal nodes, sometimes called sink nodes. There are two types of terminal nodes, the one-sink, represented as 1 and the zero-sink, represented as 0. When a BDD is not representing a constant function, it contains exactly two terminal nodes [BRB90].
Decision nodes are labelled by a variable representing the corresponding input from the Boolean function. Each decision node has two outgoing edges, the low edge and the high edge. The path from the low-edge of a decision node to another node is equivalent to assigning the truth value 0 to the respective variable in the Boolean function. Same with the high-edge assigning a 1 [BRB90].

In Figure 2.2 a representation of the NAND function as a BDD is shown. As in the binary decision tree from Figure 2.1, we use dashed lines for the low edged and solid ones for the high edges. The shown BDD in Figure 2.2 has two important properties, it is ordered and reduced. If a BDD is both ordered and reduced we call it a reduced ordered binary decision diagram [Knu09].

### 2.3.1 Ordered Binary Decision Diagrams

A BDD is considered as ordered if the different variables all appear in the same order independent of the path we take throughout the BDD starting from the root. We do not have to start with $x_1$ as the root of a BDD, every other top variable would also be a valid root node. The order in which the top variables are traversed throughout the BDD is called variable order. For each ordered BDD, there exist $n!$ different variable orders, with $n$ being the number of unique variables of the given BDD. The chosen variable order of a BDD might have noticeable effects on the complexity of the BDD as shown in Section 2.3.1 [BRB90].

There are two approaches on when to order the variables of a BDD algorithmically. Static variable ordering is done before the construction of the BDD has begun. Dynamic variable ordering is done during the construction of a BDD.

---

2This image is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license. No changes were made to the original image. The image was created by Dirk Beyer. https://creativecommons.org/licenses/by-sa/3.0/deed.en
2.3.2 Reduced Ordered Binary Decision Diagrams

In order to prevent the encoding of redundant information, we reduce a BDD to its so-called reduced form, applying the following two rules exhaustively:

(R1) Remove every node whose two children are identical and redirect the ingoing edges of the removed node to the former child of the removed node.

(R2) If two subgraphs are identical, merge them.

The correctness of those two rules can be observed in Figure 2.4 (proof by picture) or Sieling et al [Sie98], where it is formally proven. Also, the reduction of a BDD is more useful and far less complex if it is obeying a variable order. A reduced and ordered BDD is unique for each variable order. For a reduced and ordered BDD, SAT can be solved in constant time, since such a BDD collapses down to the o-sink for any not satisfiable Boolean function. As we cannot make many meaningful statements about general BDDs, we always assume the BDDs considered in this thesis are reduced and ordered unless it is stated otherwise.

2.4 Data on the Heap and Stack

In order to use data efficiently when developing software, we have to consider where to store the data we are using. In general, there are two places where data can be stored by the program, the heap and the stack. Here, we only talk about the heap and the stack in a Rust sense [KN18], since other programming languages
Figure 2.4: A diagram showing the application of reduction rules \((R1)\) and \((R2)\) on a BDD.

could potentially use a different concept than described here. In Chapter 4 on Page 17 we discuss Rust in more detail.

The stack can only contain data with limited size and is local to a function. If a function gets called, a stack frame is created where all the local variables are stored and the memory for those variables gets allocated. After leaving the scope of the function, the stack frame gets deallocated automatically and the used memory is free again. The stack is described as very fast \([\text{KN}18]\) because the size of all the variables is known ahead of time, hence it is easy to allocate memory. For the same reason, it is fast to free the memory again.

Although the stack offers many advantages, there are important things the stack is not able to do. The two most important restrictions for our purpose are:

1. The stack cannot hold data with unknown size.
2. Every thread contains its own stack.

How a list data structure for \(i32\) (32-bit signed integer in Rust) could look like, is displayed in Listing 2.1 on the next page. The problem arising with this data structure is that the size cannot be determined at compile time. The number of occurring \texttt{next} in the list is unknown at compile-time, therefore it is impossible to create a stack frame for this structure.

A BDD is a data structure that requires this behaviour. Hence, it is necessary to find a way to create recursively, or self-referencing, data structures. We achieve such behaviour by allocating the data structure on the heap and only store a pointer to the memory in the struct. Since a pointer is a data type with a fixed size we can define lists like displayed in the example in Listing 2.2 on the facing page.
2.4. Data on the Heap and Stack

Listing 2.1: A code example for failing unlimited size data structure.

```rust
struct ListElement {
    val: i32,
    next: Option<List>
} // ERROR
```

Listing 2.2: A code example for the usage of heap allocated memory in a ListElement.

```rust
struct ListElement {
    val: i32,
    next: Box<Option<List>>
}
```

By wrapping the next list elements into a type called `Box`, which is just a pointer for heap allocation, we can use the heap instead of the stack for this data structure. `Box` is the most basic heap allocation pointer Rust has to offer [KN18]. It is called `Box` because the process of allocating a stack-allocated value on the heap is called `boxing`.

The second reason we need heap allocation is to share data across multiple threads. In this case, stack allocation is not possible, because every thread got its own stack. The previously shown `Box` pointer is also not capable of doing thread-safe heap allocation because of the way the ownership (explained in more detail in Chapter 4 on Page 17) of boxes work. For thread-safe sharing we need a more sophisticated version of heap allocation called atomic reference-counting pointers which are explained in more detail in Chapter 4 on Page 17.
2 Background
3. Library Design

Designing a software architecture requires careful planning to provide all the requirements specified beforehand. The final design of this library requires to be extensible, maintainable, and parallelizable. To achieve extensibility we took inspiration from architectures with the same core concept of transforming an input format to the desired output format. The main idea behind our architecture came from the OpenGL rendering pipeline [Gro21] and the Python machine learning library scikit-learn [PVG+11] to a lesser extent, where fixed modules along the pipeline take the current data as input and transform it into the desired output.

To achieve maintainability, extensibility, and parallelism we use common and well-understood design patterns which make it easier for future developers to pick up work on the library [FW20]. To enhance maintainability and extensibility, the architecture should consist of self-contained modules which execute work in an encapsulated manner to avoid global state and remove dependencies between the modules. Data should also only flow in one direction to ensure the order of operation. Also, a manager object should provide a clean interface to the pipeline to hide the underlying complexity. Parallelism is achieved via the thread pool pattern in combination with a work-stealing queue. Because they are more implementation-dependent we discuss them in greater detail in Chapter 4 on Page 17.

In Figure 3.1 on the following page, the structure of the architecture is outlined. The first step is reading the user input via an input parser and transforming it into a common format that can be shared between the stages of the pipeline (intermediate representation). Afterwards, in the second step, we should be able to apply a static variable ordering algorithm to the intermediate representation. In the third and fourth steps, the dynamic variable ordering algorithm gets applied during the construction of the BDD. Those steps are combined because it is not possible to divide them in an obvious way. After the construction of the BDD, the manager ob-
ject holds the BDD and allows the user to execute BDD-related queries on it, like SAT and #SAT.

![Diagram](image)

Figure 3.1: A diagram for the pipeline-like structure of the library.

### 3.1 Applied Software Design Patterns

Software engineering patterns are solution templates for commonly occurring problems in software engineering [GHJV93]. They help to develop a solution for general problems, hence they can be applied without depending on a specific implementation language. We can divide the patterns into two groups, the architectural patterns, and the design patterns. Architectural patterns describe the layout of the whole architecture of a software product, whereas design patterns describe solutions to problems that occur during the development of the architecture.

After analyzing the requirements for the architecture, we concluded that the pipes and filters [GH21] pattern fulfils our needs of extensibility and maintainability to the most extend. In the pipes and filters pattern, the architecture of the software mimics a pipeline in the sense that data flows (pipes) into the beginning of the pipeline, then gets transformed by successive execution of modifying computations (filters), and as a final step, produces the desired output. This kind of architecture is comparable to an assembly line in a factory where raw material gets refined at every step of the assembly line until the finished product is assembled at the end. Because every stage is self-contained and free of side effects, connecting them to a pipeline is also comparable to function composition [ASPS96]. The pipes and filters pattern also supports our main goals of an extensible and maintainable architecture. Extensibility of the architecture is achieved because every filter of the pipeline is exchangeable by another implementation for the same filter and it is also possible to add new filters to the pipeline if necessary. The foundation for maintainability is laid by the pipes and filters pattern because every filter of the pipeline is a self-contained execution step without any global side effects. Hence, any change made inside a filter does not influence the behaviour of another part of the pipeline. Another, not yet used, advantage of this pattern is the possibility to execute multiple pipelines in parallel without extra modification to the architecture.

A common way to access the construction process in BDD libraries is to provide a manager object which in turn provides the user with a clean and easy to use interface [Jan03]. This abstraction of the underlying complexity is done via the façade pattern (sometimes called manager pattern). The façade pattern is one of the
well-known twenty-three design patterns by the Gang of Four [GHJV93]. Using the facade pattern minimizes the expert knowledge needed to use the library because the manager object can provide an interface that starts the BDD construction process with only one method call (implementation details in Chapter 4 on Page 17). It is also possible to provide more than one manager object to the user, which increases the extensibility of the library. For example, the current implementation provides two manager objects, one for sequential and for parallel execution of the BDD construction process. Additionally, it is also possible to bypass the manager object for users with expert knowledge to construct a BDD from the Boolean functions exposed by the Bryant API displayed in Table 4.2 on Page 19. This is done as shown in Section 4.1.3 on Page 19.

A design pattern to support modularity is the builder pattern, also a well-known design pattern by the Gang of Four [GHJV93]. This pattern allows us to customize the construction process of an object by specifying which components and properties the resulting object should contain. The builder pattern is especially useful in Rust because of the absence of method overloading, otherwise, we would need to provide multiple constructors with different names, which would decrease maintainability. In this architecture, the builder pattern is used to construct a facade object which internally stores what algorithms should be used at the different stages of the pipeline.

Planning architectures with parallelism in mind is a relatively recent art. This makes it even more important to hold on to design patterns to achieve good maintainability. The thread pool pattern combined with a work-stealing queue is a way to schedule many tasks on multiple executing threads via a queue. In a thread pool, a given number of threads are spawned at the beginning of the program. Threads in a thread pool do not get destroyed after they finished their work. Hence, we avoid latency based on spawning and destroying threads, especially for ones with otherwise short lifetimes.

Another pattern to consider in the future is the adapter pattern which is originally used to provide a way for two incompatible interfaces to work together. Currently, there is no need for this pattern, hence it is not used. But because of the extensibility of the architecture, it could be implemented in the future to ensure a working data flow through the pipeline.

## 3.2 Modular Components

In this section, we discuss all modular components of the architecture and what their tasks are. As described above and displayed in Figure 3.1 on the facing page, the architecture contains a pipeline with three modular stages and a manager wrapping the pipeline:
1. Input Parser
2. Static Variable Ordering Algorithm
3. Dynamic Variable Ordering Algorithm / BDD Creation
4. Manager Object

The input parser is the entry point of the pipeline, it consumes a given input in text form and transforms it into the intermediate representation of a Boolean function. This stage allows the extensibility of the BDD library by adding support for new input formats. Currently, only DIMACS CNF [DIMo8] is supported. The implementation details of this stage can be found in Chapter 4 on Page 17. A DIMACS CNF file obeys the following rules [DIMo8]:

1. The file may begin with any number of comment lines. A comment line has to start with the lower case letter c.
2. After the initial comment lines the file contains a single problem line, starting with the lower case letter p. The p is followed by the problem type, which is cnf, in the case of the DIMACS CNF file, following the number of variables, followed by the number of clauses.
3. Every line that is not marked with c or p represents a clause.
4. A clause consists of numbers, each is the variable index (starting by 1) it represents. If the number is negative it describes a negated variable.
5. Every clause has to be terminated by a final value of 0.
6. The file ends after the last clause.

Additionally, there are some special cases:

1. Variables should be numbered from 1 to N, or from 1 + M to N + M.
2. The last clause does not have to be 0 terminated.
3. A file can contain comments that are not at the beginning of the file.

The example DIMACS CNF file from the original source [DIMo8] is displayed in Listing 3.1 on the facing page, which represents the Boolean function \((x_1 \lor \neg x_3) \land (x_2 \lor x_3 \lor \neg x_1)\).

The static variable ordering algorithm is the second stage of the pipeline. It is purely optional and applies a static ordering algorithm to the intermediate representation. This stage allows extensibility of the BDD library by adding support for
new static variable ordering algorithms. This stage receives the intermediate representation, transforms it and passes the modified intermediate representation to the next stage in the pipeline. Currently, only FORCE [AMS03] is supported. FORCE is a fast and easy-to-implement static variable ordering algorithm [AMS03], which is the main reason we used this algorithm as part of our implementation.

The dynamic variable ordering algorithm and the construction of the BDD are deeply entangled and there is no obvious way to separate them [Rud93]. To still give the user of the library the possibility to use dynamic variable ordering and to make the library extensible in this stage, the library should provide the user with programming hooks to take action at specific points during the construction process of the BDD. During the planning, we discussed a programming hook whenever the hash map of a variable has to change the size, but it is not implemented for reasons stated in Chapter 4 on Page 17. This is also a suboptimal solution because it does not guarantee that all possible dynamic variable ordering algorithms are easily implementable in this architecture. There is the possibility a dynamic variable ordering algorithm also requires a separate implementation of the BDD construction process. The current state of the architecture allows the implementation of the sifting algorithm proposed by Richard Rudell [Rud93]. After this stage is finished the pipeline returns the constructed BDD.

The manager object is not part of the pipeline, instead, it contains and manages the pipeline. During the construction process of the manager object, we specify what algorithms we want to use in the pipeline. After the manager is constructed, we can instruct it to create a BDD from a given input format. Subsequent to the construction process, we can query the manager with BDD related operations, like SAT and #SAT. In the current implementation of the architecture, it is not possible to construct arbitrary managers, because there is only one algorithm implemented for each stage of the pipeline. The current implementation provides the user with a parallel executing manager and a sequential one. The implementation of the manager is explained in more detail in Chapter 4 on Page 17. After the construction process, the pipeline returns the resulting BDD and the user is able to do further analysis on the BDD, for example by applying functions from the Bryant API listed in Table 4.2 on Page 19.
3.3 Summary

In this chapter, we first stated the intention and idea of the architecture and provided an outline of the architecture in a diagram. Then, we described what patterns in software engineering are. Afterwards, we discussed the fundamental architectural pattern used to design the architecture. Then, we listed design patterns that help us implement the architecture and combined with the architectural pattern, keep the architecture extensible and maintainable. We also provided a pattern that could potentially be used in the future. Then, we listed all the modular components of the architecture to provide an overview of which parts of the architecture have to work together.
4. Implementation of the Library

In this chapter, we discuss the different techniques and technologies used to implement the prototype of the library. First, we examine the reasons to choose Rust as the programming language and give an overview of the used techniques. Then, we discuss the implementation in more detail based on code snippets.

4.1 Implementation Basics

In this section, we discuss language-independent implementation details we used to implement the library prototype. Starting with the way hash tables work in Section 4.1.1, followed by the Boolean function API proposed by Bryant in Section 4.1.2 on the next page, finally we discuss the if-then-else function in Section 4.1.3 on Page 19.

4.1.1 Hash Tables

A hash table is a data structure that stores *key-value pairs*. When storing an entry in the hash table, the key from the key-value pair gets hashed with a hash function. In the back end of the hash table lies a random access data structure, for example, an array. Those data structures cannot be of infinite size. Mostly, they are smaller than the output range of the given hash function. Therefore, we divide the hash by the size of our random access data structure and use the resulting rest as an index. This is also known as the *modulo function*.

As depicted in Figure 4.1 "BDD" and "Truth Table" both have the same hash value when using the hypothetical hash function \( h(x) \). Here, we could potentially encounter a hash collision. After we apply the modulo function, we restrict the output even more, thus we could potentially encounter a hash collision, again.
Table 4.1: A table containing the average and worst case complexities of hash tables.

<table>
<thead>
<tr>
<th>Function</th>
<th>Average Complexity</th>
<th>Worst Case Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Insert</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Delete</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

In the example in Figure 4.1 every field in the random access data structure is a so-called **bucket**, if multiple entries get assigned to the same bucket, the hash collision is resolved by appending new elements to a linked list, in short: we do not store single values in every cell of the random access data structure, we store linked lists that hold the values of the key-value pair [Knu98].

![Figure 4.1: An overview of the anatomy of a hash table.](image)

As shown in Table 4.1, the worst-case complexity is similar to the worst-case complexity of a linked list. The worst-case for the example occurs when every entry got the same index and we are stuck with a single linked list in only one cell of the random access data structure. Then why are hash tables so important? The big advantage of hash-based data structures comes from the average, or best-case scenarios where not every entry of the hash table produces a hash collision [Knu98].

We can use hash tables to store the results of our calculations and retrieve them in the future if we have to repeat the same calculation again. This technique is called **memoization** and is heavily used in this library.

### 4.1.2 General Boolean Function API by Bryant

Bryant proposed a Boolean function API to manipulate Boolean functions [Bry18]. In the prototype, a subset of the API is implemented because the full API was not necessary to build a proof of concept implementation of the proposed architecture. The missing functions can be implemented without any changes to the existing implementations. Furthermore, the examining functions are changed to not return
Table 4.2: The adjusted boolean function API as proposed by Bryant.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Functions</td>
<td></td>
</tr>
<tr>
<td>Const($b$)</td>
<td>0 or 1</td>
</tr>
<tr>
<td>Var($i$)</td>
<td>$x_i$</td>
</tr>
<tr>
<td>Algebraic Operations</td>
<td></td>
</tr>
<tr>
<td>Not($f$)</td>
<td>$\bar{f}$</td>
</tr>
<tr>
<td>And($f$, $g$)</td>
<td>$f \land g$</td>
</tr>
<tr>
<td>Or($f$, $g$)</td>
<td>$f \lor g$</td>
</tr>
<tr>
<td>Examining Functions</td>
<td></td>
</tr>
<tr>
<td>Satisfiable($f$)</td>
<td>0 if $f$ is not satisfiable, 1 otherwise</td>
</tr>
<tr>
<td>SatCount($f$)</td>
<td>Number of variable assignments which satisfy $f$</td>
</tr>
</tbody>
</table>

variable assignments, instead, they return either true or false in case of the satisfiable function and a number in case of the satcount function.

With the API listed in Table 4.2 we can create, manipulate and examine BDDs.

We only implemented a subset of the original API proposed by Bryant, since it was only necessary to create a BDD from a Boolean function in conjunctive normal form and to solve SAT and \#SAT for the created BDD.

4.1.3 If-Then-Else

The core of the library [BRB90] forms the so called If-Then-Else function, ITE in short. ITE is a function which takes three inputs $F, G, H$ and returns $G$ if $F$ evaluates to 1, otherwise $H$. We can define this function in its logical form:

$$\text{ITE}(F, G, H) = (F \land G) \lor (\neg F \land H).$$

It is known that there are multiple functions that have an attribute called functional completeness, which means every other Boolean function can be created, using only this function. So why is ITE proposed as a suitable building block for a BDD library by Bryant et al [BRB90]? The branching of the function into a then and else part is an advantage for the BDD data structure, because the data structure itself is build up the same way [BRB90, Bry86].

In Listing 4.1 on the next page we present the pseudo code for the ITE function. Since ITE is a recursive function, the first step in the pseudo code is checking for the following terminal cases:

$$\text{ite}(1, F, G) = \text{ite}(0, G, F) = \text{ite}(F, 1, 0) = F.$$
4 Implementation of the Library

ite(F, G, H) {
    if (terminal cases) {
        return result;
    } else if (computed_table has entry {F,G,H}) {
        return result;
    } else {
        let v be the associated variable of {F, G, H};
        T = ite(Fo, Go, Ho);
        E = ite(Fz, Gz, Hz);
        if (T == E) {
            return T;
        }
        R = find_or_add_unique_table(v, T, E);
        insert_computed_table({F, G, H}, R);
        return R;
    }
}

Listing 4.1: A pseudo code implementation for the ITE function.

Those terminal cases can be extended by a further one ite(F, 0, 1) = \( \bar{F} \). Lines 2 and 3 in Listing 4.1.

If the input is not in the form of the terminal cases, it will fail the terminal case check. Then we use the look-up part of the memoization technique to check if the current input got already calculated in the past, Lines 4 and 5 in Listing 4.1. If so, we return the result from a hash map where we store all the previously computed results called \textit{computed_table}. The \textit{computed_table} maps three nodes \( F, G \) and \( H \) to the result node \textit{ite}(F, G, H).

If there is no result stored yet, we calculate the result of the ITE function in a recursive fashion, Lines 7 to 9 in Listing 4.1. The associated variable of a set of nodes is the smallest associated variable of those nodes [BRB90].

Due to the Shannon decomposition we can evaluate \( F, G \) and \( H \) at 1, denoted as \( F_0, G_0 \) and \( H_0 \), and at 0, denoted as \( F_z, G_z \) and \( H_z \). The evaluation is trivial [BRB90], let \( F = (w, T, E) \), finding the cofactors of \( F \) is determined as follows:

\[
F_0 = F \text{ if } v < w, \text{ else } T \\
F_z = F \text{ if } v < w, \text{ else } E.
\]

After we calculated the results for \( T \) and \( E \) we check if both are the same, this is done to apply reduction rule (R2), from Section 2.3.2 on Page 7, directly, Lines 10 and 11 in Listing 4.1.

Then, we use the memoization technique again in the so-called \textit{unique_table} which is also a hash map, Line 13 in Listing 4.1. The \textit{unique_table} stores every node of the BDD exactly once and therefore ensures the canonical form of the represented BDD.
### Table 4.3: Every two-variable Boolean function written in terms of the ITE function.

<table>
<thead>
<tr>
<th>Boolean Function</th>
<th>ITE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>AND(F, G)</td>
<td>ite(F, G, 0)</td>
</tr>
<tr>
<td>F &gt; G</td>
<td>ite(F, NOT(G), 0)</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F &lt; G</td>
<td>ite(F, 0, G)</td>
</tr>
<tr>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>XOR(F, G)</td>
<td>ite(F, NOT(G), G)</td>
</tr>
<tr>
<td>OR(F, G)</td>
<td>ite(F, 1, G)</td>
</tr>
<tr>
<td>NOR(F, G)</td>
<td>ite(F, 0, NOT(G))</td>
</tr>
<tr>
<td>XNOR(F, G)</td>
<td>ite(F, G, NOT(G))</td>
</tr>
<tr>
<td>NOT(G)</td>
<td>ite(G, 0, 1)</td>
</tr>
<tr>
<td>F ≥ G</td>
<td>ite(F, 1, G)</td>
</tr>
<tr>
<td>NOT(F)</td>
<td>ite(F, 0, 1)</td>
</tr>
<tr>
<td>F ≤ G</td>
<td>ite(F, G, 1)</td>
</tr>
<tr>
<td>NAND(F, G)</td>
<td>ite(F, NOT(G), 1)</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
</tr>
</tbody>
</table>

ITE is amongst other functions proposed as a building block for the library, as it is possible to construct every two-variable Boolean function from the ITE function, as displayed in Table 4.3. The ITE function is used to implement the API described in Table 4.2 on Page 19.

To get a better understanding of the construction of a BDD with the ITE function we build the BDD for the NAND ($\neg(x_0 \land x_1)$) function from the ground up as an example. The variables $x_0$ and $x_1$ are just BDDs whose high edge is connected to 1 and the low edge is connected to 0, we call them $A$ and $B$. Then, to apply the AND function, we use the ITE function with $C = \text{ite}(A, B, 0)$, as shown in Figure 4.2 on the following page.

Lastly, to get the NAND function out of $C$ we apply the NOT function to it, which reads $\text{ite}(C, 0, 1)$ and is shown in Figure 4.3 on the next page. The final result is a BDD that represents the NAND function.
Rust Implementation Details

The implementation of the proposed architecture was done in the Rust programming language. We chose the language for multiple reasons. Because the library has to execute calculations in parallel, we need a language that not only supports parallelism but also was designed for parallelism \([\text{KN}18]\). We also need a language that generates fast programs, so we can avoid slowing down the whole construction process of a BDD just by our choice of language. Most other BDD libraries are written in C/C++, since Rust is as fast as C/C++ \([\text{KN}18]\) it is a good language of choice for a BDD library. Rust also provides a tool to manage dependencies and the compilation step called Cargo \([\text{KN}18]\). Rust provides us with multiple language concepts which help us write better parallel code. Those features are mutability, ownership and borrowing, which will be discussed in detail in the following.

4.2 Mutability

In Rust all variables are immutable by default, which pushes the developer in a direction to take advantage of the memory safety and concurrency Rust offers \([\text{KN}18]\).

The example in Listing 4.2 on the facing page shows the default immutability of Rust variables. A compilation error occurs when we try to compile this code snippet because \(x\) is not explicitly declared as mutable with the \texttt{mut} keyword. In Listing 4.3 on the next page we present the fixed code with the inserted \texttt{mut} keyword.
4.2 Rust Implementation Details

```rust
let x = 42;
println!("x is {}", x);
x = 82; // ERROR!
println!("x is {}", x);
```

Listing 4.2: A code example for Rust immutability.

```rust
let mut x = 42;
println!("x is {}", x);
x = 82; // This works because of the ‘mut’ keyword
println!("x is {}", x);
```

Listing 4.3: A code example for Rust mutability.

### 4.2.2 Ownership and Borrowing

In Rust, every variable binding has a property called ownership, the following explanation of the Rust programming language and its features are taken from the publicly available Rust Book [KN18].

```rust
fn foo() {
    let x = vec![1, 2, 3];
}
```

Listing 4.4: A code example for Rust ownership.

In Listing 4.4 the function foo binds a vector to the variable name x, which means x is not the owner of the vector. After x leaves scope, the vector on the stack and all the heap-allocated memory of its elements get freed. This process always takes place at the end of the scope and always in a deterministic fashion [KN18]. We can also move ownership of this vector, as seen in Listing 4.5 on the next page.

In Listing 4.5 on the following page the value is moved from x to y. Every access to x after the value is moved results in an error, because Rust ensures that exactly one valid binding to any given value exists at the same time. We used a vector for this example, because basic data types in Rust implement a behaviour called the Copy trait, a trait is comparable to interfaces in other languages. A value of a Copy type gets copied every time it gets bind to a variable.

Creating a function that takes a vector as an argument requires implementation as displayed in Listing 4.6 on the next page. Here, the function takes ownership of the vector, thus we have to return the vector in order to return the ownership again. To prevent functions written in Rust to take ownership of every value they receive we can not only move ownership, we can also borrow values. In Listing 4.7 on the following page we show how to borrow a value to a function.
Because of those different states a variable can take on in Rust, there are four ways a variable can be passed to a function, those different states are shown in Listing 4.8 on the next page.

Borrowing is the main feature of Rust and it is the reason Rust programs are not garbage collected, but still memory safe, which means there is no possibility of invalid memory reads and writes. Borrowing consists of two rules [KN18]:

1. Any borrow must last for a scope no greater than that of the owner.

2. You may have one or the other of these two kinds of borrows, but not both at the same time:
   (a) one or more references (&T) to a resource
   (b) exactly one mutable reference (&mut T)

These rules are enforced at compile time by the rust compiler. The unit in charge of checking those rules is called the borrow checker. As long as the code obeys these rules enforced by the borrow checker the resulting program is memory safe.
4.2. Rust Implementation Details

```rust
fn take_ownership_immutable(a: Vec<i32>) {
    //...
}
fn borrow_immutable(a: &Vec<i32>) {
    //...
}
fn take_ownership_mutable(mut a: Vec<i32>) {
    //...
}
fn borrow_mutable(a: &mut Vec<i32>) {
    //...
}
```

Listing 4.8: Examples for function signatures for each of the different ownership states in Rust.

### 4.2.3 LLVM Backend

Rust is a system-level language. However, it does not compile directly to machine code. Rust programs are compiled to a so-called language-independent intermediate representation, which is then fed into the LLVM backend [Lat08], a set of compilers. Then, LLVM can target nearly every platform imaginable. The output of LLVM is highly optimized machine code for the specified platform.

This is a welcomed feature for our purpose. For example, if the platform specific optimizations speed up the processing of a single node even by a very small margin, we achieve measurable performance increases for big BDDs.

### 4.2.4 Parallelization

The parallelization of the construction process takes place in the recursive branches of the ITE function, displayed in Listing 4.1 on Page 20, line 8 and 9. The idea behind the parallelization process is to spawn a new thread every time ITE has to be called recursively. We decided to use an approach called work-stealing to parallelize the construction of a BDD. We implemented this behaviour with the help of a crate called rayon.

Work-stealing is implemented the following in rayon: There is always a pool of threads available, called worker threads. Those threads are idling until they get work assigned. Rayon exposes the `join(a, b)` function, where `a` and `b` are tasks. If we call this function the first time, worker threads will start to execute task `a` and `b`. But if `join(a, b)` is called from within a worker thread $W$, then $W$ places task `b` into its work queue, announcing that it got excess work for idling threads to do, afterwards $W$ will start executing `a`. The stealing part of work-stealing comes into play when $W$ is still busy executing `a` and another worker thread takes `b` out of the queue from $W$. After $W$ finished executing `a`, it checks its own queue if there is `b` left to execute, if so, $W$ executes `b`, if not, $W$ tries to steal work from other worker threads queues [JS20].
4.2.5 Rust Standard Hash Map

The hash map Rust provides in its standard library is tailored for generalized use. Our use case of a hash map is more of a special case. Since the standard hash map provides a cryptographically secure hash function and a type of collision resolution called Robin Hood hashing, which is an extension of the open hashing technique, we had to make changes to the hash function used by the hash map.

The default behaviour of the Rust standard hash map when being shared across multiple threads is also not to our advantage. There is no good way to share the hash map across multiple threads without packing it into a type called Mutex [KN18]. A mutex protects the data it contains from multiple accesses at the same time by locking as soon as a thread accesses the data inside the mutex. After packing the complete hash map in a mutex we potentially lose a big portion of the performance gained by parallelism because threads have to wait on each other again.

Our BDD library calls the hash function of its memoization tables many times, hence it is of the highest importance to have a fast hash function. Here, we do not value cryptographically secure hashes, and we do not care about collisions that much. So we decided to use the so-called Fowler-Noll-Vo hash function (FNV) as the hash function for the hash maps [GF19] because it is designed for fast hash table and checksum use [GF19].

The Rust standard hash map opposed the previously discussed problems and therefore it would be an advantage to build a new custom hash map data structure that allows the mutex behaviour on an entry level of the hash map. To create this data structure great knowledge about unsafe Rust is necessary and the time it takes to learn the required techniques to ensure the memory safety of the resulting data structure is not in the scope of this thesis. This topic will be explained in greater detail in Chapter 7 on Page 47.

4.2.6 (Atomic) Reference Counted Smart Pointer

In this implementation of a BDD library, it is necessary for some values to have multiple owners. Multiple ownerships occur because multiple nodes of a BDD potentially point to a single node. To make this compliant with the borrow checker the Rust standard library provides us with a data structure called reference counted smart pointer, or Rc<T> in short [KN18].

Whenever a Rc is created, it is initialized with a counter starting by 1 and the value the Rc points to is allocated on the heap. Every time a Rc gets cloned this counter is incremented by 1. Whenever a Rc is dropped somewhere, the counter is decremented by 1. As soon as the Rc counter reaches 0, the value wrapped by the Rc is dropped from the heap [KN18].

In Listing 4.9 on the facing page we can see how Rc is like a wrapper around another type, in this case it is of type Rc<Vec<i32>>. Rcs are inherently memory
unsafe, and they are implemented in an unsafe \([\text{KN18}]\) fashion. But, because they are in the Rust standard library we can assume they are well tested and safe for our usage \([\text{KN18}]\).

```rust
let x = Rc::new(vec![1, 2, 3]); //create Rc pointing at a vector.
let y = Rc::clone(x); //clone the Rc, NOT the vector.
```

Listing 4.9: A code example for the usage of Rc.

One big disadvantage of Rcs is that we can only use them in a single-threaded scenario. To share Rcs over multiple threads we have to use so-called atomic reference counted smart pointers, Arc in short. Arcs and Rcs are used exactly the same, as shown in Listing 4.10, with the main difference that accessing an Arc is more expensive because it ensures thread safety and therefore introduces some overhead \([\text{KN18}]\).

```rust
let x = Arc::new(vec![1, 2, 3]); //create Arc to a vector.
let y = Arc::clone(x); //clone the Arc, NOT the vector.
```

Listing 4.10: A code example for the usage of Arc.

### 4.3 The Rust Ecosystem

The Rust programming language also contains an ecosystem manager called Cargo. Cargo contains multiple tools and commands to help managing Rust projects. To maintain this project we used the following cargo commands: cargo run, cargo build, cargo test, cargo bench.

- **cargo run** fetches all dependencies listed in the project description file called `cargo.toml`, builds the project and then runs it.
- **cargo build** works similar to cargo run but it does not run the project after building it.
- **cargo test** runs all test functions in the project and gives an overview over all successful, ignored and failed test cases.

Although cargo bench exists, we do not use it directly in our implementation, because it requires the nightly version of the Rust compiler which we do not use for this library. Instead, we are using the Criterion crate \([\text{Hei14a}]\) for benchmarking purposes. However it is still invoked via cargo bench. A crate in Rust is like an external library in other languages. Criterion is a development dependency, this it will not appear in the published version of the library, only during development, to help with the code performance analysis. Criterion does the benchmarking process in four phases. First, in the warm-up phase, the routine is executed repeatedly for a given time to fill CPU and operating system caches \([\text{Hei14b}]\). The second phase
is called the *measurement phase* where the routine is executed repeatedly and the execution times are recorded [Hei14b]. Next, is the *analysis phase* which takes the results of the *measurement phase* and, based on the results, calculates some meaningful statistics like average runtime, outliers and so on [Hei14b]. The last phase is the *comparison phase* in which the results of the *analysis phase* are compared to the stored measurements of the last run. As a result we can examine if the code improved or regressed in performance [Hei14b].

Nightly is one of three release channels of the Rust programming language. Other channels are the stable and beta release channel. The nightly releases updates every night and it includes every new commit on the Rust master branch. Six weeks after the first nightly release a new beta is released which includes all previous nightly releases. Six weeks after the last beta release there a new stable release is published. The chance of having unstable features in the stable version [KN18] can be minimized this way.

### 4.3.1 Testing

A correct implementation of the proposed architecture and library is the foundation to reason about the result of the thesis. Hence, there is no way of guaranteeing the absence of bugs in a software product, there are solutions to reduce the possibility of bugs.

Using Rust as the programming language to implement the library ensures the absence of bugs in the memory management, assumed the rust compiler is bug-free (which it probably is not \(^1\)). Because we only used safe Rust, the stable compiler and basic data structures, like hash maps and vectors, there should not be any compiler bug resulting in our library to break.

To minimize the likelihood of incorrect algorithm implementations, we have opted for test-driven development (TTD). In TTD we first create test cases to describe the behaviour for the function we want to implement. To ensure the lowest possibility for bugs we have to be as exhaustive as possible with edge cases for the given function. As an example, some test cases and the implementation for the factorial function are shown in Listing 4.11 on the facing page. To test the `fac` function we wrote test cases for the edge cases where \(n\) equals 0 and where the input would cause a wrong result because the return value of `fac` would not fit into an `u32` (unsigned 32-bit integer) anymore. We do not have to test for negative input values because the data type `u32` only allows positive values, therefore Rust would not even let us compile the code if we tried to calculate the factorial of a negative number. Lastly, we write some tests for the function to ensure the results for non-edge cases are correct as well.

We did TDD for most of the code in the prototype. As a final result we got > 80% of code coverage. This means over 80% of the written lines of code are run by

---

\(^1\)https://github.com/rust-lang/rust/issues?q=is%3Aissue+is%3Aopen+is%3Abug
// a (bad) implementation of the factorial function
fn fac(n: u32) -> u32 {
    if n == 0 {
        1
    } else {
        n * fac(n - 1)
    }
}

// omitted the setup for a unit test module
// for better readability
#[test]
fn test_fac_small() {
    assert_eq!(fac(5), 120); // tests small calculation
}

#[test]
fn test_fac_big() {
    assert_eq!(fac(12), 479001600); // tests big calculation
}

#[test]
fn test_fac_zero() {
    assert_eq!(fac(0), 1); // tests input 0
}

#[test]
#[should_panic]
fn test_fac_overflow() {
    fac(100); // should crash because of overflow
}

Listing 4.11: An implementation of the factorial function with test cases in Rust.

At least one implemented test case. Most of the untested code is in the command line interface tool. We calculated this number via a development dependency crate called Tarpaulin [McK21].

Because the implementation of a BDD library is more complex than an implementation of the factorial function, we had to take results from known BDDs and compared the results of our implementation against them.

### 4.3.2 Dependencies

To have more control on which code exactly runs in the library, we tried to use as few dependencies as possible to build this library. However, we did also not want to reinvent the wheel for every functionality, especially if there is a crate out there that handles the given problem. As a result of this, OBDDimal directly depends on the following crates:
1. clap v2.33.3
2. fnv v1.0.7
3. rand v0.8.3
4. rayon v1.5.0

Clap [PKS20] is a command line parser, we used to improved the usability of our command line tool.

Fnv [Dev17] is an implementation of the Fowler-Noll-Vo hash function. We used this function as a non cryptographical hash function for our hash maps.

Rand [Tea21] is an implementation of a random number generator. We used it to implement the FORCE heuristic.

Rayon [JS20] is an implementation of a work stealing queue. We used it to parallelize the ITE function.

There is also the dependency on the criterion [Hei14a] crate for benchmarking and the tarpaulin [Mck21] crate for test coverage, but those are a development dependencies and not part of the final product.

### 4.4 Program Structure

In this section, we examine the final program structure and compare it to the described architecture. For the exact implementation, the source code is available in the official GitHub [Net21] repository. Important code snippets and function signatures will be shown as listings in the corresponding subsection of this thesis.

Whenever something refers to a element of the source code or a source file it is emphasized in the mono text style.

#### 4.4.1 Parser

First, we look at the parser stage of the pipeline. This stage is implemented in the parser.rs and the boolean_function.rs files. The most important function in parser.rs is shown in Listing 4.12 on the next page.

This function takes a single input string, which represents the DIMACS CNF file, and produces the corresponding Cnf struct as an output. The Cnf struct is defined as shown in Listing 4.12 on the facing page. Then, the output is wrapped in a Result type because it is possible for the parser to fail on a malformed input file. Parser-Settings consists of a collection of flags, to change which rules of the DIMACS CNF should be enforced on the input file. If we want to read any other input file which

\[\text{commit hash cae505877a399c4e675bbdde1dbfccd93bb6f8c8}\]
is not of the DIMACS CNF format, we would just have to write a parser for it, and additionally, create a struct representation for it.

After retrieving the struct representation of the input file we can transform the terms of the Cnf into the intermediate representation for Boolean functions. The intermediate representation is defined in Listing 4.13.

We note that a BooleanFunction is an Operator, a left-hand side and a right-hand side which are by themselves of type Box<Symbol>. Remember, they are wrapped in a Box because the size of the BooleanFunction is not known at compile time so we have to allocate the memory on the heap, as described in Chapter 4 on Page 17. A Symbol is either a Postterminal, which represents a not negated terminal value, a Negterminal, which represents a negated terminal value or another Function, which allows this format to build any Boolean function by recursion. The Operator

```rust
pub struct Cnf {
    pub variable_count: u32,
    pub term_count: u32,
    pub terms: Vec<Vec<i32>>,
    pub order: Vec<i32>,
}

pub fn parse_string(
    input: &str,
    settings: ParserSettings) -> Result<Cnf, DataFormatError>
```

Listing 4.12: An overview of important definitions in parser.rs.

```rust
pub enum Symbol {
    Postterminal(u32),
    Negterminal(u32),
    Function(BooleanFunction),
}

pub struct BooleanFunction {
    pub op: Operator,
    pub lhs: Box<Symbol>,
    pub rhs: Box<Symbol>,
}

pub enum Operator {
    And,
    Or,
}

pub fn new_from_cnf_formula(inp: Vec<Vec<i32>>) -> Symbol
```

Listing 4.13: An overview of important definitions in boolean_function.rs.
is either an And or an Or. To represent a CNF only two operators are needed: AND and OR, because NOT [DiMo8] is encoded in the Symbol.

The function `new_from.cnf_formula` takes the terms of the Cnf, created earlier, and returns the intermediate representation as the Symbol data type.

### 4.4.2 Static Variable Ordering

After the input format was passed through the parsing pipeline we apply an optional static variable ordering algorithm.

```rust
pub enum StaticOrdering {
    NONE,
    FORCE,
}

pub fn apply_heuristic(cnf: Cnf, heuristic: StaticOrdering) -> Cnf
```

Listing 4.14: An overview of important definitions in `static_ordering.rs`.

To apply a heuristic, we just pass the Cnf created earlier into the heuristic. Then, the function returns us a new Cnf with the updated contents. This is not optimal, however, it saved development time, in Chapter 7 on Page 47 we discuss how to improve this function.

In this implementation, it is possible to choose between NONE or the FORCE heuristic for static variable ordering. To extend the static variable ordering heuristics we just have to add the name to the `StaticOrdering` enum and write a function that applies the heuristic to the given variable order. The actual algorithm that computes the new variable ordering via force can be found in `static_ordering.rs`.

### 4.4.3 Construction of the BDD and Dynamic Variable Ordering

To give the user a clean interface to the construction functions, the library exposes the functions via a manager struct.

In Listing 4.15 on the facing page we see the Manager trait and two manager structs. The trait implements every method a manager implementation has to offer. The BddManager is such an implementation, the BddParaManager offers support for parallelized execution of the build process for the BDD.

For caching purposes, a BDD manager struct holds option types for its values. As soon as the sat_count or the node_count is calculated once for the contained Bdd the value is saved in the corresponding field of the struct. Then, the value is read from the corresponding field if we access the node_count and sat_count multiple times.
### 4.4. Program Structure

```rust
pub trait Manager {...

pub struct BddManager {
    bdd: Option<Bdd>,
    sat_count: Option<u64>,
    node_count: Option<u64>,
}

pub struct BddParaManager {
    bdd: Option<Bdd>,
    sat_count: Option<u64>,
    node_count: Option<u64>,
    unique_table: Option<fnv::FnvHashMap<UniqueKey, Arc<NodeType>>>,
}
```

Listing 4.15: An overview of important definitions in `bdd_manager.rs`.

In **Listing 4.16 on the next page** we present all the methods described by the `Manager` trait. The getters described at the top are needed to give default implementations for the methods, the `new` function returns an empty manager, and the `add_bdd` function adds a given `Bdd` to the empty manager. The `from_format` function is an alternative constructor for the manager. It takes an input `string`, a `format` and `parser-settings` to decide which parser to choose. After executing, the `from_format` function returns a `Manager` containing the created `Bdd`. An excerpt of the documentation for the `Manager` trait can be found in **Listing 4.17 on Page 38**.

Once a `Manager` which contains a `Bdd` exists, we can query the `node_count`, `satisfiable` and `sat_count` method on the `Manager`. If there is no `Bdd` in the `Manager` the evaluation functions return a `NoBddError`. This `manager pattern` is common under BDD packages [Jan03].

#### 4.4.4 Serialization and Deserialization

Calculating a BDD from the ground up every time we want to use the results is cumbersome and uses more resources as necessary. To solve this, we introduced a serialization format to save already constructed BDDs to a file. The format works as follows:

1. Every line represents a node of the BDD,
2. the first line is the root node,
3. every line starts with an internal ID, followed by the associated variable, then followed by the ID of the low and high node,
4. the terminal nodes `0` and `1` are labeled by the hard-coded ID `0` and `1` respectively.
The serialization of the BDD described in Listing 3.1 on Page 15 is therefore serialized as displayed in Listing 4.18 on Page 39.

There exists a function in the library that is able to deserialize this format. Currently, this functionality is not working properly because of how the input file is read. This is described in more detail in Section 4.5 on the next page. This bug results in the feature that loading a BDD from a file is slower than rebuilding it.

4.4.5 The Command-Line Tool

To ease the use of the library, a command-line tool (CLI) was developed which makes the functionality of the library accessible. Whenever the tool is run without any parameters it displays all the possible configurations, such an invocation is shown in Listing 4.19 on Page 39. The only things needed to build a BDD with our prototype is how to run a command line program and a DIMACS CNF file containing the desired function.

The tool is capable of running the BDD calculations in a sequential or parallelized manner, if we set the -a flag we activate the parallelized mode. Whenever we want to benchmark the current execution time of a BDD, we can add the -t flag. Then a timer gets initialized and printed with the final result of OBDDimal. It is also possible to save a calculated BDD to a file and reload it in a future invocation of
OBDDimal. This is done via the \(-l\) and \(-s\), or \(-load\) and \(-output\) flags. To use a static variable ordering algorithm we just have to specify the \(-p\) or \(-preorder\) flag and name the algorithm. Currently only FORCE is available. The CLI tool, contrary to the library itself, is only able to construct BDDs out of a given input document, currently only in DIMACS CNF format. We can use a DIMACS CNF file by setting the \(-i\) or \(-input\) flag and specifying the path to the file. This operation invokes the from_format function, displayed in Listing 4.16 on the preceding page. As a final result the CLI tool always prints the #SAT evaluation of the constructed BDD.

4.5 Discussion

Like every non-trivial software project, the implementation of this prototype opposed difficulties during development. In this section, we discuss which problems occurred, and what we did to solve them, or what could be done to solve them.

4.5.1 Performance Measurements During Development

To evaluate if an implementation of a potential improvement did actually improve the performance of the library, it is important to use benchmarks to track the improving or regressing performance. Then, we can examine which parts of the library slow the library down to take action or at least be aware of parts where we have to investigate and improve in the future.

We measured the performance after every compilation. However, there were five iterations of the software which imposed a major performance increase as shown in Figure 4.4 on the following page.

Every measurement is executed with the BerkeleyDB DIMACS CNF file. This file represents a Boolean function with 76 terminal symbols, 141 terms and every benchmark executed #SAT on the BDD. This file was chosen for this purpose because the solution to #SAT is already known for it. Furthermore, the BerkeleyDB CNF provides enough complexity to result in non-trivial BDDs and the implementation of our prototype took long enough to generate the resulting BDD to get distinguishable time measurements. All measurements referenced in Figure 4.4 on the next page were done in sequential execution.

Since the program did not terminate, the zeroth measurement did not yield a result. At this point, the main goal of the implementation was to get correct results, not fast ones. After we established a correct implementation, we optimized the memory management of the BDD data structure by removing unnecessary memory allocations for BDD nodes. This improvement took the execution time down to 426 seconds (1), which is a considerable improvement. Afterwards, we decided to increase performance by improving the way hashing works in our hash maps. This was done by just hashing the pointer to the node, instead of hashing the node object itself (which also recursively hashed all children of the said node). This improved our execution time by 85%, down to 65 seconds (2). The next improvement
was achieved by a fixed bug, which caused the implementation of the Shannon decomposition to do unnecessary recursive calls. The bug fix improved the performance by another 82%, down to 12 seconds (3). Measurement (4) and (5) are small optimizations, we added additional base cases to the ITE function like proposed by Bryant et al. [BRB90] and we used the Fowler-Noll-Vo hash function like described in vrefch:implementation, a non-cryptographical hash function, inside the hash maps to improve performance to an execution time of 9 seconds, which is an additional performance gain of around 25%.

From our first measurable implementation to the last measurement, we could improve the overall execution time from 426 seconds down to 9 seconds on the BerkeleyDB file. This is an improvement of 98%. Nevertheless, state of the art BDD libraries like CUDD [CUD] and BuDDY [BuD] take considerably less time to solve this particular BDD. They only need a few milliseconds to execute the calculations.

### 4.5.2 The Parallelization of the BDD Construction Process

It is possible to construct BDDs in a parallelized way with the final tool, albeit the construction is orders of magnitudes slower for inputs of the size of our BerkeleyDB example in comparison to the sequential construction. The last measurement took around 8 seconds for the sequential construction and around 1000 seconds for the parallelized construction.
In this case, the core problem lies in how the standard implementation of the Rust hash map. It is not possible to share the hash map over multiple threads and only lock the hash map on an entry level. Instead, as soon as a thread wants to access the hash map, the whole data structure gets locked for every other thread, this happens because the hash map has to be shared in a Mutex which handles all the locking for us. In consequence, the parallelization only adds overhead to the calculations since all the threads have to queue up and do their work in sequence anyway.

Potentially, Mutex can be replaced by RwLock. A RwLock is a reader-writer lock which distinguishes between reading and writing access to the wrapped data structure. Therefore, it is able to lock the wrapped data structure only when writing access happens and it is able to stay unlocked as long as only reading access occurs.

### 4.5.3 Dynamic Variable Ordering

To prove the correctness of the proposed architecture we tried to implement one algorithm for every stage in the construction pipeline. For dynamic ordering we tried to implement an algorithm called variable sifting [Rud93]. Here, the main problem was again the safety features of Rust. Variable sifting relies on sharing a vector of hash maps across all threads since every variable now has a dedicated unique table. Only one table could be accessed at the same time because we had to share the vector via Mutex across the threads, which resulted in the same effect as described above. The threads queued up and could only work in sequence, which means the parallelization added a lot of overhead but yielded no performance gain.

### 4.5.4 Deserialization

The deserialization algorithm currently performs a linear search for each node of the serialized BDD to reconstruct the original BDD. This results in quadratic runtime complexity which is unfeasible, even for medium-sized BDDs. A solution to this problem is proposed in Chapter 7 on Page 47.

### 4.6 Summary

In this chapter, we first stated the language-independent implementation details. Afterwards, we described the Rust specific implementation details for the prototype and how the ecosystem of Rust works and what benefits it provides. Next, we provided the structure of the library implementation, which covers the modular components of the library and the command line tool. Finally, we discussed what problems occurred during development, what their consequences are and how to fix them.

---

3The author of this thesis learned about this synchronizing data structure after the implementation phase was over.
fn get_bdd(&self) -> &Option<Bdd>
fn get_sat_count(&self) -> &Option<u64>
fn get_node_count(&self) -> &Option<u64>
fn get_bdd_mut(&mut self) -> &mut Option<Bdd>
fn get_sat_count_mut(&mut self) -> &mut Option<u64>
fn get_node_count_mut(&mut self) -> &mut Option<u64>
fn new() -> Self
Creates an empty Manager struct.
fn from_format(
    cnf: &str,
    format: InputFormat,
    settings: ParserSettings,
    static_ordering: StaticOrdering
) -> Result<Self, DataFormatError> where
    Self: Sized,
Creates a BDD from a given input.
format is the given format of the input.
settings describe how the parser should interpret the input.
Returns a BddManager or a DataFormatError.
fn add_bdd(&mut self, bdd: Bdd) -> &mut Self
Adds a given Bdd to Manager and resets the
memoization of the sat_count and node_count.
fn deserialize_bdd(&mut self, input: &str) -> &mut Self
Deserializes a previously serialized Bdd onto the Manager.
fn serialize_bdd(&self) -> Result<String, NoBddError>
Serializes the current hold Bdd to a String.
fn node_count(&mut self) -> Result<u64, NoBddError>
Counts the nodes of the currently hold Bdd and
returns the result Result<u64, NoBddError>.
fn satisfiable(&self) -> Result<bool, NoBddError>
Returns Ok(true) if the given Bdd represents
a function which is satisfiable.
fn sat_count(&mut self) -> Result<u64, NoBddError>
Returns the number of inputs that satisfy the
function the current Bdd is representing.

Listing 4.17: An excerpt of the Rust documentation for the Manager trait.
Listing 4.18: A serialized BDD in our custom format.

```plaintext
11,1,3,9
3,3,1,0
9,2,6,1
6,3,0,1
```

Listing 4.19: The standard output after an invocation of the OBDDimal CLI tool without parameters.
5. Related Work

In this chapter, we discuss other well-known BDD libraries and the differences to our library, and what makes our library stand out in comparison to them.

5.1 Existing BDD Libraries

Because there are many BDD libraries out there, we want to discuss only some of them. We decided to discuss BuDDy and CUDD because they are the most well-known and go-to BDD libraries [TCT21]. Furthermore, we include Sylvan because it is also a BDD library which was planned with parallelization in mind and JavaBDD because it is a pure Java implementation and a wrapper for BuDDy and CUDD at the same time.

**CUDD** stands for Colorado University Decision Diagram and it was written by Fabio Somenzi [Som20]. This library is not only capable of constructing BDDs, but it is also able to create and calculate other types of decision diagrams. CUDD supports the complete Bryant API [BRB90] to manipulate BDDs. It was released in 1995 [Jan03], 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. Geert Janssen ranked CUDD as one of the 8 good packages they researched [Jan03]. CUDD was also not planned with parallelization in mind while creating the library.

**BuDDy** is a BDD package written by Jørn Lind-Nielsen [LN21], it supports the complete Bryant API [BRB90] for BDD manipulation. BuDDy also supports dynamic variable ordering. BuDDy is written in the C programming language, but it is also wrapped in a C++ layer to make it more accessible [MW17]. BuDDy is not manager based [Jan03], which means it can only calculate one BDD in a single program execution. BuDDy also implements some sophisticated drawing functions, useful for debugging and it is possible to send the BDD output to the graph printing program
Related Work

**DOT**, to draw created BDDs. Geert Janssen ranked CUDD as one of the 8 good packages they researched [Jan03]. BUDDy supports parallel execution [LN21].

**Sylvan** is a BDD package written by Tom van Dijk [vv15] which was designed with parallelization in mind, although it does not support dynamic variable ordering. It implements the Bryant API, but in a way that it supports parallel execution of these operations [TvD19, vDLvDP13]. Sylvan also 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 [TvD19]. Sylvan is actively developed and maintained. Unfortunately, Sylvan was not ranked by Geert Janssen [Jan03], but Sylvan was ranked by Tom van Dijk et al. [vDHJ+15] and they showed that parallelized BDD construction can outperform sequential construction if the circumstances are right (overall long construction times where multiple CPU cores can be used).

**JavaBDD** is a BDD package written by John Whaley [Wha21] which is not only a pure Java implementation, but also a wrapper for BUDDy, CUDD and more BDD libraries via the Java Native Interface. The interface for the Java implementation is the same as for the wrapper around BUDDy and CUDD, so it is easy to exchange them [Wha21].

The most distinct feature of our library in comparison to the previously mentioned ones is that there is no expert knowledge about BDDs required to construct a BDD and solve SAT for it. Because our library is able to parse input files, in its most basic case, it only requires one function call to get from an input file to a query-able BDD. Another feature of our implementation is the possibility to extend the library in a modern programming language without exactly knowing how every detail of the implementation works. This is due to the pipeline like architecture described in Chapter 3 on Page 11. It is also possible to easily configure which algorithms to use for which task in the construction process of the BDD.

### 5.2 Possible Future Extensions

Our library supports a very basic BDD data structure. Bryant et al. described some extensions for the basic BDD implementation to decrease the resulting BDD size or increase the construction speed for the BDD [BRB90].

An extension described by Bryant et al. is called complement edges [BRB90]. Complement edges are a way to exploit the fact that a node $G$ and a node $\overline{G}$ are the same except their sink nodes $0$ and $1$ are interchanged. We can mark the outgoing edges of a node using a complement bit to indicate if an edge is a complement edge. As a result, we can store a node $G$ and a node $\overline{G}$ as the same node in the unique table, if we want to represent the negated version of a node we have to point with a complement edge to it. We also do not need two terminal nodes anymore because we can represent $0$ as a complement edge to $1$, or vice versa. With complement edges, it is possible to represent the same BDD in different ways, which violates the requirement of a canonical form. Bryant et al. solved this by enforcing the rule that

---

the then edge of every node has to be a regular edge [BRB90]. For the examples by Bryant et al., introducing complement edges reduced the size of the resulting BDDs by around 7\% and the construction time decreased almost by a factor of 2, however, the examples used by Bryant et al. where very small [BRB90].

Another extension also described by Bryant et al. are standard triples [BRB90]. This technique is able to reduce the storage capacity we need for the computed table by using the fact that parameters $G_1, G_2, G_3$ may exist such that $\text{ite}(F_1, F_2, F_3) = \text{ite}(G_1, G_2, G_3)$ for $F_i \neq G_i$. Hence, it is possible to only store the representative standard triple for each call to the ITE in the computed table [BRB90].

An additional way to extend the library is by using a pointerless implementation of the library as described by Geert Jannsen [Jan01]. This approach does not use nodes that are pointing to their children, it uses a central data structure to store all the relations between nodes instead.
6. Conclusion

We proposed an architecture for a modular BDD library. The architecture is based on the pipes and filters architectural pattern as described in Chapter 3 on Page 11. This modularity ensures every stage of the pipeline can be interchanged to try and measure different configurations of algorithms working together. To prove the viability of the proposed architecture we implemented an algorithm for every stage of the pipeline, except for the dynamic variable ordering for reasons discussed in Section 4.5.3 on Page 37.

All in all, the proposed architecture was able to provide a foundation for a proof of concept implementation which was able to construct BDDs from DIMACS CNF and calculate (\#)SAT. The results are verified via the known results of non-trivial examples. The intermediate representation to share data between the different stages of the pipeline works correctly and can be extended for future use cases. Parallelization, in general, is functional but parallel BDD calculations require more time to execute than sequential ones because of reasons stated in Section 4.2.5 on Page 26.

Our prototype implementation is not able to compete with state of the art implementations of existing BDD libraries and it was never meant to do so, but we compiled a collection of future improvements to the library to increase the overall performance in Chapter 7 on Page 47.

Finally, we want to emphasize that the architecture is a suitable foundation for a parallelized BDD library written in Rust, and the prototype is extensible via future work because of the applied software engineering design patterns.
7. Future Work

In this chapter, we discuss future work to expand the idea of this project and to increase its efficiency in BDD construction. We also summarize the current state of the library and the missing parts.

A major problem with the library right now is the fact that we can't share the standard hash map of Rust over multiple threads without locking it entirely as soon as one thread accesses it. This could be solved by building a custom hash map data structure for this library. It might be suitable, if this custom hash map could also implement a non-cryptographic hash function and do a hash collision resolution that is to our advantage, like creating collision chains for every entry. At the same time, we could get rid of a bug in the current implementation which results in too many nodes in the final unique table. We narrowed this bug down to the implementation of the standard hash map in Rust. However, we could not confirm this completely.

Another major problem with the library is the absence of dynamic variable ordering algorithms. We tried to implement the variable sifting algorithm but it relied on sharing multiple hash maps over several threads. Thus, we couldn't do it without slowing down the whole construction process even further. Therefore, we need to find a way to share a vector over multiple threads which can be locked entrywise. This most certainly results in a custom implementation of such a data structure.

Because we only implemented one algorithm for every stage of the processing pipeline we only use a single struct, called Cnf, to pass the values through the pipeline. However, the name of this struct is not generalized (Cnf), it has to be changed to a more sensible name if we start to use other input formats as well. It should be possible to create a Rust trait out of the properties of this struct, and allow the use of a better name for the data passed through the pipeline.

The deserialization algorithm is not needed to construct a BDD from the ground up, however, it is needed to save time for future analysis of the same BDD, because it
saves construction time. The current problem is the linear search on every node in the input file like described in Section 4.5 on Page 35. A solution to this problem would be to store every node in the file in a hash map, afterwards look up the nodes in the order they are occurring in the serialized BDD in the constructed hash map. Because hash map look ups are faster than linear search it should improve the time it takes to deserialize a BDD dramatically.
Bibliography


[Boo] George Boole. An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities. London. (cited on Page 1)


[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. 1995. (cited on Page 2)


Declaration of Authorship

I hereby declare that this thesis is my own unaided work. All direct or indirect sources used are acknowledged as references. This paper was not previously presented to another examination board and has not been published as of yet.

Place, Date of Submission ___________________________ Signature ___________________________