UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

# Toolkit for automated reasoning and interpolation with ordered resolution

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science (B.Sc.)
im Studiengang Informatik

vorgelegt von

Tim Merker

Erstgutachter: Prof. Dr. Viorica Sofronie-Stokkermans
Univ. Koblenz-Landau, Institut für Informatik

Zweitgutachter: M.Ed. Dennis Peuter
Univ. Koblenz-Landau, Institut für Informatik

Koblenz, im August 2021

# Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

| | Ja | Nein |
|---|---|---|
| Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden. | ☒ | ☐ |

. . . . . .

(Ort, Datum) (Unterschrift)

# Zusammenfassung

In dieser Arbeit wird ein Toolkit für automatisches Beweisen und Interpolation durch geordnete Resolution entwickelt. Zunächst werden die Grundlagen der Aussagenlogik erklärt. Dazu zählt die Syntax und Semantik, die Transformationen von Formeln in die konjunktive und disjunktive Normalform, die geordnete Resolution mit Redundanzeliminierung und die Craig-Interpolation. Basierend auf den theoretischen Grundlagen wird eine Reihe von Tests in einer Konsolenanwendung implementiert, welche als Toolkit benutzt werden. Diese Konsolenanwendung kann im Erklärungsmodus oder Ergebnismodus ausgeführt werden und bietet auch eine Hilfe für Studierende. Die genutzte Programmiersprache des Toolkits ist Java.

Für die Implementierung der geordneten Resolution wird die Formel als Eingabe eingelesen und in eine KNF-Formel umgewandelt, wobei der Nutzer die Wahl zwischen zwei möglichen Methoden hat. Die erste Transformation ist eine sechsschrittige Umformung in eine logisch äquivalente Formel in KNF und die zweite eine erfüllbarkeitserhaltende Transformation einer Formel in NNF zu einer Formel in KNF. Durch die geordnete Resolution wird die Klauselmenge auf Unerfüllbarkeit überprüft. Falls die Formel unerfüllbar ist, wird falsum zurückgegeben. Falls stattdessen eine Klauselmenge erfüllbar ist, wird ein Modell mit Hilfe einer kanonischen Konstruktion generiert und als Ausgabe zurückgegeben. Die geordnete Resolution wird in einem weiteren Test verwendet, um redundante Klauseln in einer Klauselmenge zu identifizieren und zu eliminieren. Es werden nur spezifizierte Atome für die Inferenzen verwendet. Diese Atome werden in ihrer Klausel als maximal betrachtet. In der Craig-Interpolation wird die geordnete Resolution erneut angewendet. Die Craig-Interpolation liefert eine Interpolante, die den Grund für die Unerfüllbarkeit zweier Klauselmengen darstellt.

# Summary

In this bachelor thesis, we developed a toolkit for automated reasoning and interpolation with ordered resolution. Firstly, the theoretical aspects of propositional logic are covered in the preliminaries section, including syntax and semantics, the transformation of formulae in conjunctive and disjunctive normal form, the ordered resolution calculus with redundancy elimination, and Craig interpolation. Based on this, we developed a console application which provides these functionalities. The program can be run in a teaching mode or the result mode and serves as a guideline for students in propositional logic. The program is written in the programming language Java.

For the implementation of the ordered resolution calculus, we require a formula as an input. We implemented two methods for computing CNFs, the first one using equivalences in propositional logic, the second one using structure-preserving transformations. Through ordered resolution, we test the satisfiability of a clause set. If it is unsatisfiable, we return falsum and we construct proof for the unsatisfiability of the clause. Otherwise, if the clause set is satisfiable, we return a model of the clause set using the canonical construction of interpretations. The ordered resolution calculus is used in a further test, to identify redundant clauses of a clause set and eliminate them. For the Craig interpolation, we use the ordered resolution calculus. Only specified atoms are used in the inferences of the ordered resolution. These atoms are marked as maximal in their clause. Last but not least, Craig interpolation is a further application of the ordered resolution calculus. Craig interpolation returns an interpolant, which is the cause for the unsatisfiability of the union of two clause sets.

# Contents


**1 Introduction** **1**
- 1.1 Related work . . . 2
- 1.2 Structure of the thesis . . . 3

**2 Preliminaries** **5**
- 2.1 Propositional logic . . . 5
  - 2.1.1 Syntax . . . 5
  - 2.1.2 Semantics . . . 9
- 2.2 Conjunctive normal form . . . 11
  - 2.2.1 CNF transformation using equivalences in propositional logic . . . 11
  - 2.2.2 Structure preserving CNF transformation . . . 13
- 2.3 Ordered resolution calculus . . . 14
  - 2.3.1 Soundness of resolution . . . 16
  - 2.3.2 Refutational completeness of resolution . . . 18
- 2.4 Craig interpolation . . . 22

**3 Implementation** **25**
- 3.1 Workflow . . . 25
  - 3.1.1 Workflow of the ordered resolution test . . . 26
  - 3.1.2 Workflow of the redundancy test . . . 28
  - 3.1.3 Workflow of formula transformations . . . 28
  - 3.1.4 Workflow of the computing interpolant test . . . 29
  - 3.1.5 Teaching mode and result mode . . . 30
- 3.2 Data structures . . . 32
  - 3.2.1 Data structures for literals, clauses, and clause sets . . . 34
  - 3.2.2 Ordering on clauses and clause sets . . . 36
  - 3.2.3 Data structures for representing propositional formulas . . . 39
- 3.3 Polarity of a subformula of a given formula . . . 42
- 3.4 Conversion of formulae . . . 43
  - 3.4.1 Conversion to NNF using equivalences . . . 43
  - 3.4.2 Conversion to CNF using equivalences . . . 44
  - 3.4.3 Conversion to DNF using equivalences . . . 46
  - 3.4.4 Conversion to CNF using structure preserving transformation . . . 47
- 3.5 Ordered resolution calculus . . . 50
  - 3.5.1 Factorization rule . . . 50
  - 3.5.2 Resolution rule . . . 51
  - 3.5.3 Algorithm . . . 53

3.5.4 Construction of a model . . . . . 55
3.6 Redundancy . . . . . 57
3.6.1 Redundancy of a clause w.r.t. a clause set . . . . . 57
3.6.2 Redundancy elimination . . . . . 59
3.7 Interpolation . . . . . 61

**4 Conclusion** **65**
4.1 Future Work . . . . . 65

**Bibliography** **67**

# 1 Introduction

Satisfiability (SAT), the problem of checking satisfiability of formulas in propositional logic, gained interest from researchers from various fields. Most finite problems with an underlying finite set can be formulated as the satisfiability problem for suitable propositional formula. If we can prove the satisfiability of the propositional formula and derive a model, we can retranslate the model and derive a solution. Satisfiability has many applications. Below, we mention its application in electronic circuits and in program verification. After that, we discuss different methods to test satisfiability, and conclude with interpolation.

One can establish a natural link between electric circuits and propositional logic: The input resp. output is usually a voltage which can have two values 0 (low) and 1 (high). Logic gates implement Boolean operations; thus with every electric circuit formed using logic gates one can associate in a natural way a propositional formula which describes the link between the values between the inputs and outputs.

Propositional logic is widely used in the verification of finite state systems. If we assume given a set $\Pi$ of propositional variables and a "labeling" function which indicates, for every state, which propositional variables are true, then we can represent states and sets of states as formulae over the set $\Pi$ of propositional variables. Transitions can be also represented as formulae over $\Pi \cup \Pi'$ when $\Pi$ are names for values of the propositional variables before and $\Pi'$ names for values of these variables after the transitions. There exist verification methods for finite state systems which manipulate such propositional formulae.

SAT is famous also in theoretical computer science. It is the "typical" NP-complete problem and therefore of great theoretical interest. One method to test the satisfiability of formulas is the ordered resolution calculus, which we implemented in this thesis. There are also different methods to test satisfiability: Truth tables, semantic tableau, and the Davis-Putnam-Logemann-Loveland procedure DPLL. In this thesis, we focus on the ordered resolution calculus and show its application in redundancy testing and interpolation. Elimination of redundant clauses is used to "simplify" the sets of clauses, which is very beneficial because it helps reduce the number of clauses, hence also the number of inferences which need to be performed.

Interpolation has many applications, among which we here mention a few: In modular databases, columns of databases are encoded as propositional variables. Operations like natural join are translated to propositional formulas [8]. When we have two databases given, we can prove if the logical modeling $F_1 \wedge F_2$ of the two databases is consistent. If it is not consistent, i.e. $F_1 \wedge F_2 \models \bot$, then we can apply Craig interpolation to locate the error. Craig interpolation returns an interpolant $I$ containing only propositional variables occurring in $F_1$ and $F_2$ such that $F_1 \models I$ and $I \models \neg F_2$.

Another application of Craig interpolation is again in program verification [7]. Last but not least, Craig interpolation is also used in reasoning in combinations of theories. The interpolant shows which information needs to be exchanged between two provers. As one can see, satisfiability testing and interpolation are related problems, where interpolation locates the reason for unsatisfiability.

## 1.1 Related work

Since satisfiability testing is a relevant topic in a variety of fields, researchers have already developed toolkits to solve satisfiability problems. Among the existing provers to check satisfiability in propositional logic we mention Chaff (that uses DPLL). ZChaff[1] from the Princeton University is a prover for propositional logic. It uses an algorithm that is resolved from the DPLL procedure. Since DPLL is a more efficient approach than the ordered resolution, zChaff relies on the DPLL procedure to be competitive. First order theorem provers like SPASS, E, and Vampire can in theory also be solved for checking satisfiability of propositional formulae. SPASS[2] developed at the Max Planck Institute for Informatics, is an automated theorem prover for first-order logic which uses the superposition calculus, which is a version of ordered resolution with built-in equality handling. E[3] developed in Stuttgart is another theorem prover for full first-order logic with equality which has acquired its reputation as a fast theorem prover. Andrei Voronkov's Vampire[4] uses the ordered binary resolution and superposition for handling equality. It is known to be very efficient. SPASS, E, and Vampire are mainly designed for first-order logic and strive for efficiency.
Our toolkit takes a different approach. We want to provide tests for propositional logic that are based on the ordered resolution calculus and make them as beginner-friendly as possible. The ordered resolution calculus is not the most efficient algorithm for checking satisfiability, but in contrast to the other calculi of the SAT solvers, the ordered resolution calculus can be applied for redundancy testing and Craig interpolation. Our toolkit shows these applications of the ordered resolution calculus, and beginners learning propositional logic can see how an implementation of the ordered resolution calculus and its applications can look like. By having two program modes, one can choose whether he wants to run the toolkit in the educational mode or the result mode. The educational mode is supported with additional explanations, which will help to understand the methods behind the ordered resolution calculus. The result mode is kept to return the plain results. Focussing on the ordered resolution calculus and on another audience makes this toolkit different from the other provers.

[1] `https://www.princeton.edu/~chaff/zchaff.html`
[2] `https://www.mpi-inf.mpg.de/departments/automation-of-logic/software/spass-workbench`
[3] `https://wwwlehre.dhbw-stuttgart.de/~sschulz/E/E.html`
[4] `http://www.vprover.org/index.cgi`

## 1.2 Structure of the thesis

The thesis is structured as follows: In Chapter 2, we cover the theoretical aspects of propositional logic. The preliminaries chapter starts with the basics of propositional logic, describes the transformation of formulae in conjunctive normal form, defines the ordered resolution calculus, and concludes with its application in redundancy testing and Craig interpolation. In Chapter 3, we present the workflow of the toolkit, the teaching and the result mode, the data structure behind these functionalities, and of course all the functions which are provided by the toolkit. In Chapter 4, we conclude this thesis with a summarization of the work, my experiences working on the toolkit and a discussion of possible future extensions for the toolkit.
The toolkit is public and can be accessed via the SVN version control system. To gain access, please contact my supervisor Prof. Dr. Viorica Sofronie-Stokkermans[5] via email.

[5] sofronie@uni-koblenz.de

# 2 Preliminaries

## 2.1 Propositional logic

In this section, we introduce the main notations on propositional logic which are important for the thesis. Propositional logic can be used for encoding and reasoning about information about finite domains when certain properties can be "true" or "false". To describe a logic we have to describe its syntax and its semantics. The syntax describes which are the well-formed formulas. The semantics defines the "meaning" or interpretation of formulas. In this section, the syntax and semantics are described as well as the terms validity, satisfiability, and unsatisfiability.

### 2.1.1 Syntax

To define syntax of propostional logic we use propositional variables and the logical connectives $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$. [10]

Let $\Pi$ be a set of *propositional variables*. We use letters $P$, $Q$, $R$, $S$ to denote propositional variables.

It is also possible to subscript integers to the letters for more variety, e.g. $P_1$, $P_2$, $P_3$.

**Definition 2.1 (Propositional formulas [10]).** *$F_\Pi$ is the smallest set with the following properties:*

*1)* $\top, \bot$ *and all propostional variables are in* $F_\Pi$.

*2) If* $F, G$ *are in* $F_\Pi$ *then*

*2.1)* $\neg F$,

*2.2)* $(F \wedge G)$,

*2.3)* $(F \vee G)$,

*2.4)* $(F \rightarrow G)$,

*2.5)* $(F \leftrightarrow G)$

*are in* $F_\Pi$.

A formula $F = P$, when $P \in \Pi$ is also called an *atom*. The logical connectors $\neg, \wedge, \vee, \rightarrow$ and $\leftrightarrow$ are ranked by the strength of their precedence, which is indicated in Table 2.1. Parentheses can be left out if an operator with higher precedence connects formulas with lower precedence. Thus, we assume that $\neg$ binds stronger than $\wedge$, $\wedge$ stronger than $\vee$, $\vee$ stronger than $\rightarrow$, and $\rightarrow$ stronger than $\leftrightarrow$. In Table 2.1 you can see the total order of the operators and their respective precedence. The greater the integer, the higher the precedence.

**Table 2.1:** Precedence of the operators.

| Precedence | Operator |
|---|---|
| 5 | $\neg$ |
| 4 | $\wedge$ |
| 3 | $\vee$ |
| 2 | $\rightarrow$ |
| 1 | $\leftrightarrow$ |

**Example 2.2 (Precedence of a formula).** *For example, $\wedge$ binds stronger than $\vee$, so in the formula $(F \wedge G) \vee H$ parenthesis can be left out like this: $F \wedge G \vee H$.*

**Definition 2.3 (Position of a subformula in a formula [10]).** *A position is a word over $\{1, 2\}$. To adress a specific subformula of $F$ at position $p \in pos(F)$ we use the following recursive definition:*

$$
\begin{aligned}
F|_\epsilon &:= F \\
(\neg F)|_{1p} &:= F|_p \\
(F_1 \circ F_2)|_{ip} &:= F_i|_p \text{ where } i \in \{1,2\},\ p \in \{1,2\}^* \text{ and } \circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}
\end{aligned}
$$

**Example 2.4 (Position of a subformula in a formula).** *Consider the following formula: $F := (P \vee \neg Q) \rightarrow (P \leftrightarrow (\neg\neg P \wedge Q))$. We want to extract the specific subformula at position 2211 from $F$.*

$$
\begin{aligned}
F|_\epsilon &= (P \vee \neg Q) \rightarrow (P \leftrightarrow (\neg\neg P \wedge Q)) \\
F_\epsilon|_2 &= (P \leftrightarrow (\neg\neg P \wedge Q)) \\
F_2|_2 &= (\neg\neg P \wedge Q) \\
F_{22}|_1 &= \neg\neg P \\
F_{221}|_1 &= \neg P
\end{aligned}
$$

*$\neg P$ is the subformula of $F$ at position 2211 .*

**Definition 2.5 (Polarities [10]).** *The polarity of a subformula $G$ of a formula $F$ describes the number of negations starting from $F$ down to $G$. The polarity of the subformula is 1 if the number is even, -1 if the number is odd and 0 if there is at least one equivalence connective along the path. In order to determine the polarity of a subformula, we also need the correct position of the subformula in the formula. Note that subformulas can appear several times in a formula at different positions. The polarity of a subformula*

*$G = F|_p$ at position $p$ is $pol(F, p)$, where $pol$ is recursively defined by:*

$$\begin{aligned} pol(F, \epsilon) &:= 1 \\ pol(\neg F, 1p) &:= -pol(F, p) \\ pol(F_1 \wedge F_2, ip) &:= pol(F_i, p) \\ pol(F_1 \vee F_2, ip) &:= pol(F_i, p) \\ pol(F_1 \rightarrow F_2, 1p) &:= -pol(F_1, p) \\ pol(F_1 \rightarrow F_2, 2p) &:= pol(F_2, p) \\ pol(F_1 \leftrightarrow F_2, ip) &:= 0 \end{aligned}$$

**Example 2.6 (Polarity of a given formula).** *We give two different examples. Firstly, let $F = (Q \wedge \neg P) \vee (\neg Q \rightarrow \neg R)$. The polarities of the respective subformulas $pol(F, p)$ are as follows:*

$$\begin{aligned} pol(F, \epsilon) &= pol(F, 1) \\ &= pol(F, 2) \\ &= pol(F, 11) \\ &= pol(F, 12) \\ &= pol(F, 22) \\ &= pol(F, 211) \\ &= 1 \end{aligned}$$

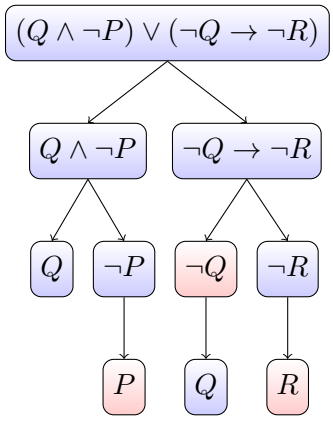
*As one can see the subformula at position 211 is the second occurrence of $Q$, which occurs under an even number of negations.*

$$\begin{aligned} pol(F, 121) &= pol(F, 21) \\ &= pol(F, 221) \\ &= -1 \end{aligned}$$

*The formula at position 121 is $P$ and occurs under an odd number of negations. The computation of the polarity can also be computed by generating a tree for the formula. As one can see in Table 2.2 below. This is also the approach we will take in the implementation of the polarity computation in Section 3.3 and explain in more detail.*

**Table 2.2: First tree based polarity computation.**

Blue for positive polarity, red for negative polarity.




**Example 2.7 (Polarity of a given formula).** *Consider now the following formula* $H = \neg\neg Q \wedge ((P \wedge Q) \leftrightarrow (P \vee Q))$.
*Below you can see the polarities of all subformulas of* $H$ *at position* $p$.

$$\begin{aligned} pol(H, \epsilon) &= pol(H, 1) \\ &= pol(H, 2) \\ &= pol(H, 111) \\ &= 1 \end{aligned}$$

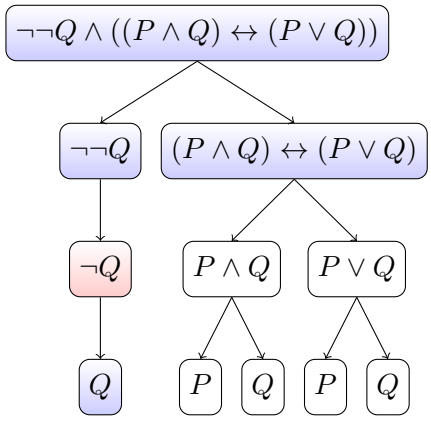
$$pol(H, 11) = -1$$

$$\begin{aligned} pol(H, 21) &= pol(H, 22) \\ &= pol(H, 211) \\ &= pol(H, 212) \\ &= pol(H, 221) \\ &= pol(H, 222) \\ &= 0 \end{aligned}$$

*The tree according to the given example can be found in Table 2.3.*

**Table 2.3: Second tree based polarity computation.**

Blue for positive polarity, red for negative polarity, white for zero polarity.




### 2.1.2 Semantics

The semantics of propositional logic defines the "meaning" of the formulas. The first step is to provide a way of evaluating propositional variables. In propositional logic, this is achieved by "evaluating" the formulas as true (1) or false (0).

**Definition 2.8 (Valuations [10]).** *Propositional variables can be evaluated to truth values by a valuation. $\mathcal{A}$ $\Pi$-valuation is a map*

$$\mathcal{A} : \Pi \to \{0, 1\},$$

*where $\{0, 1\}$ is the set of truth values.*

The value of a propositional variable in a valuation can be 0 or 1, but not both at the same time. Valuations can be extended to formulas as follows:

**Definition 2.9 (Truth value of a formula in $\mathcal{A}$ [10]).** *The evaluation of a formula $\mathcal{A}^* : F_\Pi \to \{0, 1\}$ is defined inductively as follows:*

$$\begin{aligned}
\mathcal{A}^*(\bot) &= 0 \\
\mathcal{A}^*(\top) &= 1 \\
\mathcal{A}^*(P) &= \mathcal{A}(P) \\
\mathcal{A}^*(\neg F) &= 1 - \mathcal{A}^*(F) \\
\mathcal{A}^*(F \wedge G) &= min(\mathcal{A}^*(F), \mathcal{A}^*(G)) \\
\mathcal{A}^*(F \vee G) &= max(\mathcal{A}^*(F), \mathcal{A}^*(G)) \\
\mathcal{A}^*(F \to G) &= max(1 - \mathcal{A}^*(F), \mathcal{A}^*(G)) \\
\mathcal{A}^*(F \leftrightarrow G) &= \textit{if } \mathcal{A}^*(F) = \mathcal{A}^*(G) \textit{ then } 1 \textit{ else } 0
\end{aligned}$$

**Example 2.10 (Evaluation of a formula).** *Consider the formula* $\neg F \to G$. *The evaluation of the formula* $\mathcal{A}^*(\neg F \to G)$ *is:*

$$\begin{aligned}\mathcal{A}^*(\neg F \to G) &= max(1 - \mathcal{A}^*(\neg F), \mathcal{A}^*(G)) \\ &= max(1 - (1 - \mathcal{A}^*(F)), \mathcal{A}^*(G)) \\ &= max(1 - (1 - \mathcal{A}(F)), \mathcal{A}(G)) \\ &= max(1 - 1 + \mathcal{A}(F), \mathcal{A}(G)) \\ &= max(\mathcal{A}(F), \mathcal{A}(G))\end{aligned}$$

*The formula* $\mathcal{A}^*(\neg F \to G)$ *is true if either* $F = 1$ *or* $G = 1$.

**Models, Validity, and Satisfiability** We now define notations such as the model of a formula, as well as satisfiability and validity. In this paragraph, we want to introduce these terms and provide a proper definition.

**Definition 2.11 (Models [10]).** *Let* $F$ *be a* $\Pi$*-formula. We say that* $F$ *is true under* $\mathcal{A}$ *(*$\mathcal{A}$ *is a model of* $F$*;* $F$ *is valid in* $\mathcal{A}$*;* $F$ *holds under* $\mathcal{A}$*), written* $\mathcal{A} \models F$*, if* $\mathcal{A}(F) = 1$. *We say that* $F$ *is valid or that* $F$ *is a tautology, written* $\models F$*, if* $\mathcal{A} \models F$ *for all* $\Pi$*-valuations* $\mathcal{A}$. $F$ *is called satisfiable if there exists an* $\mathcal{A}$ *such that* $\mathcal{A} \models F$. *Otherwise* $F$ *is called unsatisfiable (or contradictory).*

**Validity vs. Unsatisfiability** Checking validity goes hand in hand with checking unsatisfiability. If a formula $F$ is valid, then $F$ is automatically not unsatisfiable. Instead, if a formula $F$ is unsatisfiable, then there is no model $\mathcal{A}$ for $F$ such that $\mathcal{A} \models F$.

**Lemma 2.12 (Negation of a valid formula [10]).** $F$ *is valid if and only if* $\neg F$ *is unsatisfiable.*

*Proof.* ($\Rightarrow$) If $F$ is valid, then $\mathcal{A}(F) = 1$ for every valuation $\mathcal{A}$. This concludes that $\mathcal{A}(\neg F) = 1 - \mathcal{A}(F) = 0$ for every valuation $\mathcal{A}$, so $\neg F$ is unsatisfiable. [10]

($\Leftarrow$) However, if $\neg F$ is unsatisfiable, then $\mathcal{A}(\neg F) = 0$ for every valuation $\mathcal{A}$. This means that $\mathcal{A}(F) = 1 - \mathcal{A}(\neg F) = 1$ for every valuation $\mathcal{A}$, so $F$ is valid. □

Some methods like the resolution calculus or tableau calculi only check satisfiability. If we want to use such methods to check the validity of a formula $F$, we can reduce checking validity of a formula $F$ to negating $F$ and checking whether $\neg F$ is unsatisfiable.

**Checking Unsatisfiability** Let $F$ be a formula. As $F$ contains a finite number of propositional variables, it is possible to generate every possible valuation and check if $F$ is satisfiable in that valuation or not.
The number of possible valuations depends on the number $n$ of propositional variables in $F$. Every propositional variable can take two values 0 or 1. Hence, the number of possible valuations is $2^n$.
Therefore, the time complexity of the algorithm is also bad, up to $O(2^n)$. SAT is the problem of deciding, given a propositional formula whether the formula is satisfiable. SAT is

an NP-complete problem. Firstly, this means that SAT is in NP, that is the complexity class where decision problems can be solved in polynomial time by a non-deterministic Turing machine. And secondly it means, that every problem in NP can be reduced to the SAT problem in polynomial time. It is not known whether NP = P (where P is the class of problems that can be solved deterministically in polynomial time). If an algorithm was found that solves the SAT problem in polynomial-time, then the complexity class NP would equal the complexity class P, because SAT is NP-complete. During calculi for checking satisfiability, we might avoid testing all valuations. (See Definition 2.31)

## 2.2 Conjunctive normal form

A formula is in *conjunctive normal form* if each *clause* is connected by a disjunction. A clause consists of *literals*, that are connected by a disjunction. Finally, literals are positive or negative propositional variables.

**Definition 2.13 (CNF [10]).** *Let $C_1, ...C_n$ be clauses. Within each clause a finite number of literals is connected by disjunctions. A conjunctive normal form is defined as follows:*

$$\bigwedge_{i=1}^{0} C_i = \top$$
$$\bigwedge_{i=1}^{1} C_i = C_1$$
$$\bigwedge_{i=1}^{n} C_i = \bigwedge_{i=1}^{n-1} C_i \wedge C_n$$

There are two methods to systematically generate for every formula an equivalent formula in CNF. These methods are studied in Subsection 2.2.1 and Subsection 2.2.2 .

### 2.2.1 CNF transformation using equivalences in propositional logic

The given algorithm to convert any formula to CNF consists of five steps. [10]
Let $H[F]_p$ be a formula in which $F$ occurs as a subformula at position $p$. In each step the rules are applied until no further application is possible.

Step 1: Eliminate equivalences:

$$H[F \leftrightarrow G]_p \Rightarrow_{CNF} H[(F \rightarrow G) \wedge (G \rightarrow F)]_p$$

Step 2: Eliminate implications:

$$H[F \rightarrow G]_p \Rightarrow_{CNF} H[\neg F \vee G]_p$$

Step 3: Push negations downward:

$$H[\neg(F \vee G)]_p \Rightarrow_{CNF} H[\neg F \wedge \neg G]_p$$
$$H[\neg(F \wedge G)]_p \Rightarrow_{CNF} H[\neg F \vee \neg G]_p$$

Step 4: Eliminate multiple negations:

$$H[\neg\neg F]_p \Rightarrow_{CNF} H[F]_p$$

Step 5: Push disjunctions downward:

$$H[(F \wedge F') \vee G]_p \Rightarrow_{CNF} H[(F \vee G) \wedge (F' \vee G)]_p$$

Step 6: Eliminate $\top$ and $\bot$:

$$H[F \wedge \top]_p \Rightarrow_{CNF} H[F]_p$$
$$H[F \wedge \bot]_p \Rightarrow_{CNF} H[\bot]_p$$
$$H[F \vee \top]_p \Rightarrow_{CNF} H[\top]_p$$
$$H[F \vee \bot]_p \Rightarrow_{CNF} H[F]_p$$
$$H[\neg\bot]_p \Rightarrow_{CNF} H[\top]_p$$
$$H[\neg\top]_p \Rightarrow_{CNF} H[\bot]_p$$

**Example 2.14.** *Consider the following formula* $(F \leftrightarrow \neg G) \vee \neg(\neg F \wedge G)$*:*
*Step 1: Eliminate equivalences:*

$$(F \leftrightarrow \neg G) \vee \neg(\neg F \wedge G)$$
$$\equiv((F \rightarrow \neg G) \wedge (\neg G \rightarrow F)) \vee \neg(\neg F \wedge G)$$

*Step 2: Eliminate implications:*

$$\equiv((\neg F \vee \neg G) \wedge (\neg\neg G \vee F)) \vee \neg(\neg F \wedge G)$$

*Step 3: Push negations downward:*

$$\equiv((\neg F \vee \neg G) \wedge (\neg\neg G \vee F)) \vee (\neg\neg F \vee \neg G)$$

*Step 4: Eliminate multiple negations:*

$$\equiv((\neg F \vee \neg G) \wedge (G \vee F)) \vee (F \vee \neg G)$$

*Step 5: Push disjunctions downward:*

$$\equiv((\neg F \vee \neg G \vee F \vee \neg G) \wedge (G \vee F \vee F \vee \neg G))$$
$$\equiv_{\textit{law of excluded middle}}((\top) \wedge (\top))$$

*Step 6: Eliminate $\top$ and $\bot$:*

$$\equiv \top$$

**Definition 2.15 (Negation Normal Form (NNF) [10]).** *A formula is in negation normal form (NNF) if all negations occur right before the atoms in the formula.*

In steps 1 and 2 equivalences and implications are broken down, which contain negations as can be seen in the definition of Step 2 in Subsection 2.2.1 . In Step 3 negations are pushed downward as long as they appear right before the propositional variables. Step 4 simply eliminates redundant double negations.

### 2.2.2 Structure preserving CNF transformation

To check satisfiability of a formula $F$ we can obtain a formula $G$ that is *equisatisfiable* to $F$ and check satisfiability of $G$ to conclude satisfiability of $F$. This is more efficient than using equivalences in propositional logic. $F$ and $G$ are equisatisfiable if the following condition is met: $G \models \bot$ if and only if $F \models \bot$. [10] [9]

**Definition 2.16 (Tseitin transformation [10]).** *Let $H = H[F]_p$ be a formula, where $F$ occurs in $H$ as a subformula at position $p$. $H$ is satisfiable if and only if $H[Q]_p \wedge (Q \leftrightarrow F)$ is satisfiable, where $Q$ is a new propositional variable that works as an abbreviation for $F$. The rule is used recursively for all subformulas in the original formula. The resulting conjunctions are converted to CNF.*

**Example 2.17.** *Consider the following formula $F := \neg(P \vee Q) \wedge \neg Q$. Then the following formula obtained using polarity-based transformations is equisatisfiable to $F$.*

$$H := P_\epsilon \wedge (P_\epsilon \leftrightarrow (P_1 \wedge \neg Q)) \\ \wedge (P_1 \leftrightarrow (\neg P \wedge \neg Q))$$

*Definition 2.16 assures that $H \models \bot$ if and only if $F \models \bot$.*

**Theorem 2.18 (Polarity-based CNF transformation [10]).** *Let $Q$ be a propositional variable not occuring in $H[F]_p$. Define the formula $def(H, p, Q, F)$ by*

- $(Q \rightarrow F)$, *if* $pol(H, p) = 1$,
- $(F \rightarrow Q)$, *if* $pol(H, p) = -1$,
- $(Q \leftrightarrow F)$, *if* $pol(H, p) = 0$.

*Then $H[F]_p$ is satisfiable if and only if $H[Q]_p \wedge def(H, p, Q, F)$ is satisfiable.*

**Example 2.19.** *Let $F := (P \rightarrow \neg Q) \leftrightarrow \neg(Q \rightarrow R)$ be a formula. Then the following formula obtained using polarity-based CNF transformation is equisatisfiable to F:*

$$\begin{aligned} H := P_\epsilon \wedge (P_\epsilon \rightarrow (P_1 \leftrightarrow P_2)) \\ \wedge (P_1 \leftrightarrow (P \rightarrow \neg Q)) \\ \wedge (P_2 \leftrightarrow (P_3)) \\ \wedge ((Q \leftrightarrow R) \rightarrow P_3) \end{aligned}$$

*As we can see by Definition 2.18 , H is equisatisfiable to F. ($H \models \bot$ if and only if $F \models \bot$).*

Now these formulas can be converted to CNF by applying the six steps introduced in Section 2.2.1.

## 2.3 Ordered resolution calculus

The resolution calculus takes as input a set of clauses and uses two inference rules for deriving consequences of these clauses.

**Definition 2.20 (The resolution calculus *Res* [10]).** *Let $C$ and $D$ be clauses, $A$ an atom and $L$ a literal.*

*Resolution inference rule:*

$$\frac{D \vee A \quad C \vee \neg A}{D \vee C}$$

*Factorization inference rule:*

$$\frac{C \vee L \vee L}{C \vee L}$$

*The order of the atoms in the clause set which are used for resolution or factorization is irrelevant, because disjunctions are commutative and associative.*

The resolution calculus is a calculus for checking satisfiability by systematically deriving consequences of the clauses in a clause set and checking whether falsum was derived. If the saturated clause set contains $\bot$ then $N$ is unsatisfiable. In contrast to the regular resolution calculus, the *ordered resolution*, in addition, assumes given a total and well-founded ordering $\succ_L$ on literals and uses only maximal literals in the inferences. The literal ordering $\succ_L$ can be defined starting with an order $\succ$ on propositional variables [9]. Literals ordered by $\succ_L$ have the following properties:

$$\neg P \succ_L P \text{ and if } P \succ Q \text{ then: } P \succ_L Q, \neg P \succ_L Q, P \succ_L \neg Q, \neg P \succ_L \neg Q$$

$\succ_L$ is extended to an ordering $\succ_C$ on clauses. $\succ_C = (\succ_L)_{mul}$, the *multi-set extension* of $\succ_L$. We use $\succ$ as the notation for $\succ_L$ and $\succ_C$.

**Definition 2.21 (Multi-set [6]).** *A multi-set (or bag) $M$ is a collection of objects that may contain repeated occurrences of elements and there is no particular arrangement of the elements. We use the notation $M(m)$ to indicate the number of occurrences of an element $m$ in the multi-set $M$.*

**Example 2.22 (Multi-set).** *Let $M_1$, $M_2$, and $M_3$ be multi-sets.*

$$M_1 = [A, A, B, C, C, C, C, D]$$
$$M_2 = [A, B, C, C, D]$$
$$M_3 = [A, A, B, C, C, C, C, D]$$

*Note that $M_1 \neq M_2$, as $M_1(A) = 2$ and $M_2(A) = 1$. However, $M_2$ is considered a subset of $M_1$. In fact, every set $N$ can be interpreted as a multi-set $N'$, with $\forall n \in N : N'(n) = 1$. Two multi-sets are equal, if for every element in both multi-sets the number of occurrences in both multi-sets are equal. This is the case for $M_1$ and $M_3$.*

**Definition 2.23 (Multi-set extension [1]).** *Given a strict order $>$ on a multi-set $M$, and two multi-sets $S_1$ and $S_2$. We define the corresponding multi-set order $\succ_{mul}$ over $M$ as follows:*

$$\begin{aligned} S_1 \succ_{mul} S_2 \;\Leftrightarrow\; & S_1 \neq S_2 \\ & \text{and } \forall m \in M : [S_2(m) > S_1(m) \\ & \Rightarrow \exists m' \in M : (m' \succ m \text{ and } S_1(m') > S_2(m'))] \end{aligned}$$

The literals in the clause are rearranged by the given order $\succ$. On the left the maximal literal appears and on the right the lowest. Clauses are ordered by their maximal literal.

**Example 2.24.** *Consider the following order $O := R \succ Q \succ P$ and the clause set $N$ given below:*

$$N := (\neg P \vee \neg R) \wedge (Q \vee R \vee \neg P) \wedge (Q \vee P) \wedge (\neg Q)$$

*Ordering $N$ by $O$ will result in the following permutation. We underline the maximal literal in the clauses:*

1. $\underline{Q} \vee P$
2. $\underline{\neg Q}$
3. $\underline{R} \vee Q \vee \neg P$
4. $\underline{\neg R} \vee \neg P$

**Definition 2.25 (The ordered resolution calculus $Res_\succ$ [10]).** *Let $C$ and $D$ be clauses, $A$ an atom, $L$ a literal, and $\succ$ an order on propositional variables.*

*Ordered resolution inference rule:*

$$\frac{D \vee A \quad C \vee \neg A}{D \vee C}, \text{ if } A \text{ is strictly maximal in } D \vee A \text{ w.r.t. } \succ.$$

*Ordered factorization rule:*

$$\frac{C \vee L \vee L}{C \vee L}, \text{ if } L \text{ is maximal in } C \vee L \vee L \text{ w.r.t. } \succ.$$

**Example 2.26.** *Consider the order $O := R \succ Q \succ P$ and the clause set $N$ as shown in the previous example. We apply $Res_\succ$ on $\mathbb{N}$ and underline the maximal literal in the clauses:*

$$N := (\neg P \vee \neg R) \wedge (Q \vee R \vee \neg P) \wedge (Q \vee P) \wedge (\neg Q)$$

| | | |
|---|---|---|
| ***1.*** | $\underline{Q} \vee P$ | *(given)* |
| ***2.*** | $\underline{\neg Q}$ | *(given)* |
| ***3.*** | $\underline{R} \vee Q \vee \neg P$ | *(given)* |
| ***4.*** | $\underline{\neg R} \vee \neg P$ | *(given)* |
| ***5.*** | $\underline{P}$ | *(Res. 1,2)* |
| ***6.*** | $\underline{Q} \vee \neg P \vee \neg P$ | *(Res. 3,4)* |
| ***7.*** | $\underline{\neg P} \vee \neg P$ | *(Res. 2,6)* |
| ***8.*** | $\underline{\neg P}$ | *(Fac. 7)* |
| ***9.*** | $\bot$ | *(Res. 5,8)* |

*In order to prove that the ordered resolution calculus $Res_\succ$ gives the correct answer, i.e. a set of clauses $N$ is unsatisfiable iff the empty clause $\bot$ can be derived from $N$ in $Res_\succ$, we prove that the calculus is sound and complete. We will prove in Section 2.3.1 the soundness of the calculus and completeness in Section 2.3.2. As the ordered resolution is sound and $\bot$ is inferred, the clause set $N$ is unsatisfiable.*

### 2.3.1 Soundness of resolution

Soundness of resolution guarantees that if a set $N$ of clauses is satisfiable, $\bot$ cannot be obtained from $N$ in $Res_\succ$. For the soundness of the resolution, we need to prove Lemma 2.27, because we will use it later in the proof for theorem 2.28.

**Lemma 2.27 ([9]).** *Consider the two clauses $C_1 \vee P$ and $C_2 \vee \neg P$. We prove that $(C_1 \vee P) \wedge (C_2 \vee \neg P) \models C_1 \vee C_2$.*

*Proof.* Let $\mathcal{A}$ be a valuation with $\mathcal{A}(C_1 \vee P) = 1$ and $\mathcal{A}(C_2 \vee \neg P) = 1$. In order to proove $\mathcal{A}(C_1 \vee C_2) = 1$, two cases need to be considered:

1. If $\mathcal{A}(C_1) = 1$ then $\mathcal{A}(C_1 \vee C_2) = 1$.
2. If $\mathcal{A}(C_1) = 0$ then $\mathcal{A}(P) = 1$. Therefore, $\mathcal{A}(C_2) = 1$, because $\mathcal{A}(C_2 \vee \neg P) = 1$ and $\mathcal{A}(\neg P) = 0$.

This concludes $\mathcal{A}(C_1 \vee C_2) = 1$. □

**Theorem 2.28 (Propositional resolution is sound 2 [10]).** *Let $N$ be a set of clauses: If $\bot$ can be derived from $N$ with the resolution calculus $N \vdash_{Res} \bot$, then $N$ is unsatisfiable.*

*We here give only the idea of the proof [9].*
1. We use the notation $Res^*(N) = \bigcup_{n \in \mathbb{N}} Res^n(N)$ for the set of all possible resolvents that can be derived by $Res$ in a finite number $n \in \mathbb{N}$ of steps. $C$ is in $Res^*(N)$ if $C \in \bigcup_{n \in \mathbb{N}} Res^n(N)$, which means that $C$ is a possible resolvent of $N$ by the resolution calculus at a certain step $n$. By induction using Lemma 2.27 one can prove that if we derive a clause $C$ from a clause set $N$ by the resolution calculus, then $N \equiv N \cup \{C\}$.
2. If $\bot$ can be derived from $N$ with the resolution calculus, $N \vdash_{Res} \bot$, then $\bot \in Res^*(N)$, which means that $N \equiv N \cup \{\bot\}$.
3. $N$ is satisfiable if there exists at least one valuation for $N$ which makes all clauses of $N$ satisfiable. But $N \cup \{\bot\}$ is unsatisfiable, because there exists no valuation that makes the clause $\bot$ satisfiable. Therefore $N$ is also unsatisfiable.

□

### 2.3.2 Refutational completeness of resolution

Refutational completeness of resolution means that if a clause set $N$ is unsatisfiable, $\bot$ can be derived from $N$ in a finite number of steps in $Res_{\succ}$. To prove refutational completeness of resolution, we must introduce the model existence theorem and the construction of interpretations. The model existence theorem is a major part of the resolution calculus. Whenever we cannot derive $N \models \bot$ by $Res$, the model existence theorem guarantees that a model for $N$ must exist. We will also see, from the way the "canonical model" is built in this case, we can better understand which clauses are not needed to derive a contradiction or build a model and are therefore redundant. In the construction we use productive clauses for the generation of a model and this hints to the conclusion that unproductive clauses are unnecessary. We start with some properties of the clause orderings, which we need in our proofs throughout the entire subsection.

#### 2.3.2.1 Properties of the clause orderings

By having a closer look at Definition 2.23, we can derive two major properties of the clause orderings.

**Proposition 2.29 (Properties of the Clause Ordering [10]).**
*1. The orderings on literals and clauses are total and well-founded.*
*2. Let $C$ and $D$ be clauses with $A$ being the maximal literal of $C$, and $B$ being the maximal literal of $D$.*

*(i) If $A \succ B$ then $C \succ D$.*
*(ii) If $A = B$, and $A$ occurs negatively in $C$ but only positively in $D$, then $C \succ D$.*

#### 2.3.2.2 Construction of interpretations

If we know that a clause set $N$ is satisfiable, we can use the construction of interpretations to construct a model $\mathcal{A} : \Pi \to \{0, 1\}$ for $N$. For the construction we consider instead of "preliminary" valuations $\mathcal{A}$ that are constructed, the set $I$ of propositional variables that are true for $N$. $N$ need not be saturated. However, if $N$ is saturated, the construction cannot fail for a clause which makes it easier to compute an interpretation. This is also the method that we will use in the implementation to generate a model for any given satisfiable and saturated formula.

**Definition 2.30 (Construction of interpretations [10]).** *When we have a clause set $N$ and an ordering $\succ$, we first check satisfiability by applying the resolution calculus $Res_{\succ}(N)$. If $\bot \notin N$ and $N$ is saturated, our goal is to find a set of propositional variables $I$ such that $I \models N$. The construction is based on our order $\succ$ and starts with the smallest clause.*

**Main ideas of the constrcution. We closely follow [9][10].**

- $I_C$ stands for the set of all atoms that are true in the model we construct.
- Clauses are ordered by $\succ$.
- When considering a clause $C$, one already has a partial interpretation so far available ($I_C$). Initially $I_C = \emptyset$.
- If $I_C \models C$, no atom needs to be added to $I_C$.
- If $I_C \not\models C$, the maximal atom of $C$ must be added to $I_C$, such that $C$ becomes true.
- Changes should, however, be *monotone*. An atom should never be deleted from $I_C$, and the truth values of clauses smaller than $C$ should be maintained.
- An atom $A$ is added, $\Delta_C = \{A\}$, if $C$ is false in $I_C$, $A$ occurs as a positive literal in $C$, and $A$ is the maximal literal in $C$. Otherwise, $\Delta_C = \emptyset$.
- The construction fails for a clause if $I_C \not\models C$ and $\Delta_C = \emptyset$.
- If the construction fails for a clause, then some inference with the smallest such clause (i.e. "minimal counterexample") has not been computed. Otherwise, $I_N = \bigcup \Delta_C$ with $C \in N$ is a model of all clauses. The construction should not fail for a saturated clause set.

**Definition 2.31 (Construction of candidate interpretations [10]).** *Let $N$ be a clause set and $\succ$ an ordering. We define sets $I_C$ and $\Delta_C$ for all ground clauses $C$ over the given signature inductively over $\succ$:*

$$I_C := \bigcup_{C \succ D} \Delta_D$$

$$\Delta_C := \begin{cases} \{A\}, & \text{if } C \in N, C = C' \vee A, A \succ C', I_C \not\models C \\ \emptyset, & \text{otherwise} \end{cases}$$

*The candidate interpretation for $N$ (w.r.t. $\succ$) is given as $I_N^{\succ} := \bigcup_C \Delta_C$. We simplify the notation $I_N^{\succ}$ to $I$ if $N$ and $\succ$ are clear from the context.*

**Example 2.32 (Construction of candidate interpretations).** *We construct the canonical model $I_N$ for the extended clause set $N = (\neg P_2 \vee P_3) \wedge (P_4) \wedge (P_4 \vee P_4) \wedge (P_3 \vee P_4) \wedge (\neg P_3 \vee P_1 \vee P_1) \wedge (\neg P_2 \vee P_1)$. Let $P_4 \succ P_3 \succ P_2 \succ P_1$. Firstly, we sort the clauses from smallest to largest and underline the maximal literal of each clause. We start with the smallest clause $C_1 := \underline{\neg P_2} \vee P_1$ on top. We check if $I_C \models (\underline{\neg P_2} \vee P_1)$. $I_C$ is the set of all atoms that are true under the model we construct. $I_C$ is initialized with $I_C = \emptyset$. As the literal $\neg P_2$ of clause $C_1$ appears in negative form and $P_2 \notin I_C$, we conclude that $I_C \models C_1$. We remark that $I_C$ is a model of $C_1$ corresponding to the valuation $\mathcal{A}_C$: $\{P_1, P_2, P_3, P_4\} \to \{0, 1\}$ with $\mathcal{A}(P_i) = 0$ for all $i \in \{1, 2, 3, 4\}$. No atom must be*

*added to $\Delta_C$ such that $C_1$ becomes true. Now we move on to clause $C_2 := \underline{P_3} \vee \neg P_2$. $C_2$ is also true in $\mathcal{A}_C$, because $P_2 \notin I_C$. $C_3 := \underline{\neg P_3} \vee P_1 \vee P_1$ is true in $\mathcal{A}_C$, because $P_3 \notin I_C$. $C_4 := \underline{P_4}$ is false in $\mathcal{A}_C$, because $P_4 \notin I_C$. We must add the maximal atom $P_4$ to $I_C$ in order to make $I_C$ a model of the clause $C_4$. We use $\Delta_C = \{P_4\}$ as a notation for adding the maximal atom $P_4$ in $I_C$. $C_5 := \underline{P_4} \vee P_3$ is true in $\mathcal{A}_C$, because $P_4 \in I_C$ and $I_C$ is a model of the clause $C_5$. $C_6 := \underline{P_4} \vee P_4$ is true in $\mathcal{A}_C$, because $P_4 \in I_C$ and $I_C$ is a model of the clause $C_6$.*

| | *clauses $C$* | $I_C$ | $\Delta_C$ | *Remarks* |
|---|---|---|---|---|
| *1* | $\underline{\neg P_2} \vee P_1$ | $\emptyset$ | $\emptyset$ | *true in $\mathcal{A}_C$* |
| *2* | $\underline{P_3} \vee \neg P_2$ | $\emptyset$ | $\emptyset$ | *true in $\mathcal{A}_C$* |
| *3* | $\underline{\neg P_3} \vee P_1 \vee P_1$ | $\emptyset$ | $\emptyset$ | *true in $\mathcal{A}_C$* |
| *4* | $\underline{P_4}$ | $\emptyset$ | $\{P_4\}$ | *$P_4$ is maximal* |
| *5* | $\underline{P_4} \vee P_3$ | $\{P_4\}$ | $\emptyset$ | *true in $\mathcal{A}_C$* |
| *6* | $\underline{P_4} \vee P_4$ | $\{P_4\}$ | $\emptyset$ | *true in $\mathcal{A}_C$* |

*The resulting $I_N^{\succ} = \{P_4\}$ is a model of the clause set corresponding to the valuation $\mathcal{A}$: $\{P_1, P_2, P_3, P_4\} \to \{0, 1\}$ with $\mathcal{A}(P_4) = 1$, and $\mathcal{A}(P_i) = 0$ for $i = 1, 2, 3$.*

**Remark 2.33 (Structure of $N, \succ$ [9]).** *Figure 2.1 shows the general structure of a clause set $N$ that is sorted according to $\succ$. At the top the smallest clauses according to $\succ$ are positioned and at the bottom the largest. The meaning of the notation $max(D) = B$ is that $B$ is the maximal atom of the clause $D$. A clause $C$ is productive if there exists a positive literal $A$ with $\Delta_C = \{A\}$. Figure 2.1 marks possibly productive clauses. These clauses are minimal and the maximal atom appears positive, so there is a higher chance that they are productive.*

#### 2.3.2.3 Model existence theorem

We now present the important model existence theorem, which states that for every saturated clause set $N$ with $\bot \notin N$, a model for $N$ must exist. This theorem allows us to confidently apply the construction of candidate interpretations on every saturated clause set $N$ with $\bot \notin N$ and derive a model for $N$.

**Proposition 2.34 (Model existence theorem [10]).** *Let $\succ$ be a well-founded clause ordering reduced by a literal ordering $\succ$. If $N$ is a saturated clause set w.r.t. Res and $\bot \notin N$, then for every clause $C \in N$, $C$ is productive or $I_C \models C$.*

*Proof.* Proof by contradiction. Let $N$ be saturated w.r.t. the clause ordering $\succ$ is *Res* and $\bot \notin N$. Assume that the proposition does not hold. As $N$ is not empty and well-founded, there must exist a minimal clause $C \in N$ (w.r.t. $\succ$) such that $C$ is neither productive nor $I_C \models C$. As $\bot \notin N$ then $C \neq \bot$, which means that $C$ has a maximal literal. Now we must analyse why $C$ is not productive:
**Case 1**: The maximal literal $\neg A$ is negative, i.e. $C = C' \vee \neg A$. Then $I_C \models A$, and $I_C \not\models C'$, because $I_C \not\models C$ and $C = C' \vee \neg A$ holds by assumption. This means that there

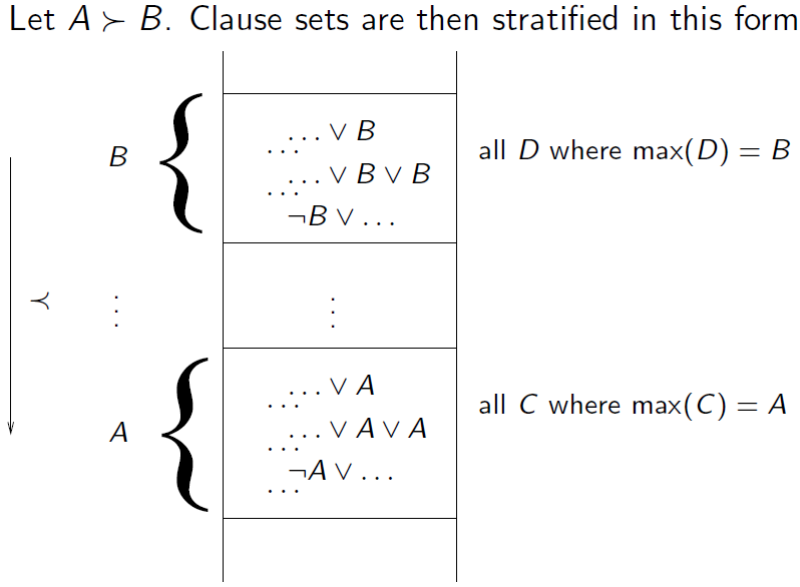



**Figure 2.1:** Structure of $N, \succ$. Figure taken from [9].

is a smaller clause $D = D' \vee A$ in $N$ with $C \succ D$, such that $D$ produces $A$ and $I_C \not\models D'$. Applying the resolution rule on $C$ and $D$

$$\frac{D' \vee A \quad C' \vee \neg A}{D' \vee C'}$$

results in $D' \vee C' \in N$. $D' \vee C'$ is smaller than $C$, because the strictly maximal atom in $C$ is $A$ and $A \notin D' \vee C'$. However, $I_C \not\models D' \vee C'$, which leads to the conclusion that $D' \vee C'$ is neither productive nor $I_{D' \vee C'} \models D' \vee C'$. This contradicts the minimality of $C$, because we can find a smaller clause than $C$ in $N$ which is neither productive nor true according to the interpretation $I_C$.
**Case 2**: Consider now that the maximal literal $A$ is positive, but not strictly maximal, i.e. $C = C' \vee A \vee A$. Applying factorization we get

$$\frac{C' \vee A \vee A}{C' \vee A}$$

and the resolvent $C' \vee A$ is smaller than $C$. As $I_C \not\models C' \vee A$, this means that the smaller interpretation $I_{C' \vee A} \not\models C' \vee A$ and does not produce $A$ either. Again, we can find a smaller clause than $C$ in $N$ and show that the smaller clause is not true according to $I_C$. This contradicts the minimality of $C$. □

#### 2.3.2.4 Refutational completeness of the ordered resolution calculus $Res_\succ$

Finally, we can use the model existence theorem and the construction of interpretations to prove the completeness of resolution like Bachmair & Ganzinger did in 1990.

**Theorem 2.35 (Refutational Completeness of Resolution (Bachmair & Ganzinger 1990) [2]).** *Let $N$ be a set of clauses and $\succ$ a clause ordering. If $N$ is saturated w.r.t. $Res_{\succ}$ and $\bot \notin N$, then $I_N^{\succ} \models N$.*

*Proof.* Firstly, we order the clauses of $N$ by $\succ$ as defined by the multiset order $\succ_{mul}$ in Definition 2.23. By Proposition 2.34 we know that if $N$ is saturated w.r.t $Res_{\succ}$ and $\bot \notin N$, then for every clause $C \in N$, $C$ is productive or $I_C \models C$. If for every clause $C \in N$, $C$ is productive or $I_C \models C$, then $I_N^{\succ} \models N$. The last part is a property of the construction that is proven in [10]. □

#### 2.3.2.5 Redundancy elimination

The goal of redundancy elimination is to delete unnecessary clauses of a clause set $N$, without changing the value of any possible valuation $\mathcal{A}(N)$. Clauses that do not produce any literal in the construction of an interpretation from Section 2.3.2.2 and are no minimal counterexamples are redundant.

**Definition 2.36 (Formal notion of redundancy [10]).** *Let $N$ be a set of ground clauses and $C$ a ground clause. $C$ is redundant w.r.t. $N$, if there exist $C_1, ..., C_n \in N$, $n \geq 0$, such that $C_i \prec C$ for all $i \in \{1, ..., n\}$ and $C_1, ..., C_n \models C$.*

**Lemma 2.37 (Redundant clauses are not productive).** *If a ground clause $C$ is redundant and all clauses smaller than $C$ hold in $I_C$, then $C$ holds in $I_C$. Assuming $C_1, ..., C_n$ are smaller than $C$, then already $C_1, ..., C_n \models C$ and $\Delta_C = \emptyset$, as $C$ is true in $I_{C_n}$. So $C$ is not productive (and neither a minimal counterexample).*

**Theorem 2.38 (Saturation up to redundancy [10]).** *Let $N$ be saturated up to redundancy. Then*

$$N \models \bot \text{ if and only if } \bot \in N.$$

*Idea of the proof [10].* If a clause $C \in N$ is redundant then by Lemma 2.37 we know that $C$ is not used as premise for "essential" inferences. As we infer $D' \vee C'$ by a resolution inference, $D' \vee C'$ is contained in $N$ or in the set of redundant clauses of $N$, $Red(N)$. If $D' \vee C'$ is in $N$, then the minimality of $C$ guarantees that $D' \vee C'$ is productive or true in $I_{D' \vee C'}$. Else, if $D' \vee C' \in Red(N)$, then $I_{D' \vee C'} \models D' \vee C'$ must still be the case, because otherwise $D' \vee C'$ would not be redundant. In both cases for the resolution rule, we get a contradiction. The same works with factorization. In [10], the proof is explained in more detail. □

## 2.4 Craig interpolation

In [5], Craig proved that first-order logic has the interpolation property. The theorem below is a specialized version for propositional logic, and a direct consequence of the soundness and the completeness of $Res_{\succ}$.

**Theorem 2.39 (Craig 1957 [10]).** *Let $F$ and $G$ be two propositional formulas such that $F \models G$. Then there exists an interpolant $H$ for $F \models G$, i.e. a formula which contains only propositional variables occuring both in $F$ and in $G$, and has the property that $F \models H$ and $H \models G$.*

*Proof [10].* Consider $\Pi_F$ the set of propositional variables which occur only in $F$ and not in $G$, $\Pi_G$ the set of propositional variables which occur only in $G$ and not in $F$ and $\Pi_{FG}$ the set of propositional variables occuring in $F$ and $G$. Convert $F$ and $\neg G$ into CNF and derive the respective clause sets $N$ and $M$. Using $Res_{\succ}$, saturate $N$ into $N'$ and after that $N' \cup M$ to derive $\bot$. Now we extract all clauses from $N'$ which only contain symbols from $\Pi_{FG}$. Connecting these clauses by conjunction returns an interpolant $H$. □

**Example 2.40.** *Consider the following propositional formulas:*

$$F := (\neg P \wedge Q) \vee (P \wedge R) \vee (\neg Q \wedge R),$$
$$G := (\neg R \wedge S) \vee (\neg S) \vee (Q).$$

*Firstly, we sort the variables of the propositional formulas into these sets:*

$$\Pi_F = \{P\},$$
$$\Pi_G = \{S\},$$
$$\Pi_{FG} = \{Q, R\}.$$

*We define an atom ordering $\succ$, such that the propositional variables in $\Pi_F$ are strictly larger than those in $\Pi_{FG} \cup \Pi_G$: $\succ := P, Q, R, S$. Thus, $\neg P \succ P \succ \neg Q \succ Q \succ \neg R \succ R \succ \neg S \succ S$. As we can tell by the resolution calculus, $F \vee G$ is unsatisfiable, which proves that an interpolant $H$ exists. Now we saturate $F$ into $F'$ by the resolution calculus w.r.t. $\succ$.*

$$F' = (R) \vee (Q \wedge R) \vee (\neg Q \wedge R) \vee (P \wedge R) \vee (\neg P \wedge Q).$$

*The interpolant consists of the disjunction of all clauses of $F'$ that only contain symbols from $\Pi_{FG}$. So $H = (R) \vee (Q \wedge R) \vee (\neg Q \wedge R)$, and after eliminating redundant clauses we can even derive a shorter interpolant $H' = (R)$.*

# 3 Implementation

We now present the actual toolkit, but give some technical details first. The programming language used for this implementation is *Java*, specifically java version 13.0.1[1]. My preferred integrated development environment for java is *IntelliJ IDEA Ultimate*[2], which can be acquired for free by the students. For testing purposes we use *JUnit 5*[3], which we added by the project management tool *Maven*[4]. Last but not least, we used *Javaluator*[5] for evaluating propositional formulas as a string. The implementation chapter is about the toolkit. In Section 3.1, we will have a look at the general workflow of the program, and the workflow of every test of the program in more detail. The workflow section concludes with the characterization of the two modes in which the program can be used, the teaching and the result mode, which the user can choose at the beginning. In Section 3.2, we present the data structure of the important units of the program, how they are implemented, and how their syntax for the toolkit is defined. In Section 3.3, we explain how the polarity of a subformula of a given formula is computed in the implementation. In Section 3.4, we present the different formula transformations and provide explanations for the tree-based algorithms that we have used to transform formulae in our program. This includes the two different methods to compute CNFs, the method to compute NNF, and the method to compute DNF. In Section 3.5, we present the implementation of the ordered resolution calculus. We present programming code for the factorization rule, the resolution rule, the algorithm for the ordered resolution calculus, and the construction of an interpretation when the clause set is satisfiable. In Section 3.6, we show how ordered resolution can be used for redundancy tests. The ordered resolution calculus is used to detect redundant clauses of a clause set. In Section 3.7, we conclude this chapter by presenting the implementation of the method for computing an interpolant, which is a further application of the ordered resolution calculus.

## 3.1 Workflow

In this section, we will present the general structure and features of the toolkit. The structure is described in Figure 3.1. The toolkit receives an input that determines the program mode, i.e. teaching mode or result mode. The input is set by the user in the console, where $e$ stands for teaching mode and $r$ stands for result mode. According to the program mode, some tests will have additional output, which is for educational purposes.

[1] `https://www.oracle.com/java/technologies/javase/jdk13-archive-downloads.html`
[2] `https://www.jetbrains.com/de-de/idea/download/`
[3] `https://junit.org/junit5/`
[4] `https://maven.apache.org/`
[5] `http://javaluator.sourceforge.net/en/home/`

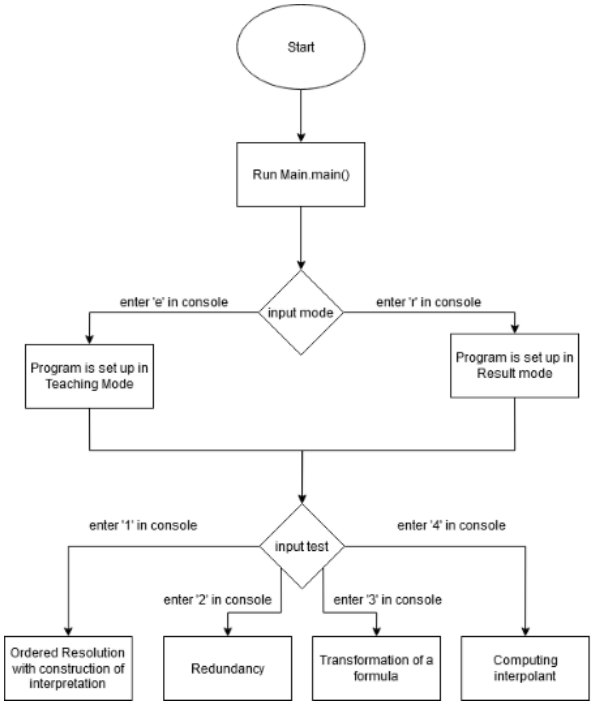



**Figure 3.1:** Workflow diagram.

We will talk more about it in Section 3.1.5. After the mode is set, the user can choose between four basic test categories.

1. Ordered resolution with construction of a model (for satisfiable clause sets),
2. Redundancy test,
3. Transformation of a formula, and
4. Computing an interpolant.

Now we will explain the four functionalities in detail.

### 3.1.1 Workflow of the ordered resolution test

The general structure of the satisfiability test based on ordered resolution can be found in Figure 3.2. Firstly, the satisfiability test based on ordered resolution receives a formula. The formula type can either be a *propositional formula* or a *clause set*. The data structure of the formula types is presented in Section 3.2. The syntax of the formula type "clause set" is explained in Section 3.2.1.3, and for the formula type "propositional formula" in Section 3.2.3. When the formula type is set to propositional formula, then an equivalent formula in CNF is computed using the method described in Section 3.4.2 and transformed into a clause set. This should improve usability, as the user does not have to first run the CNF transformation to use the ordered resolution calculus. The ordered resolution

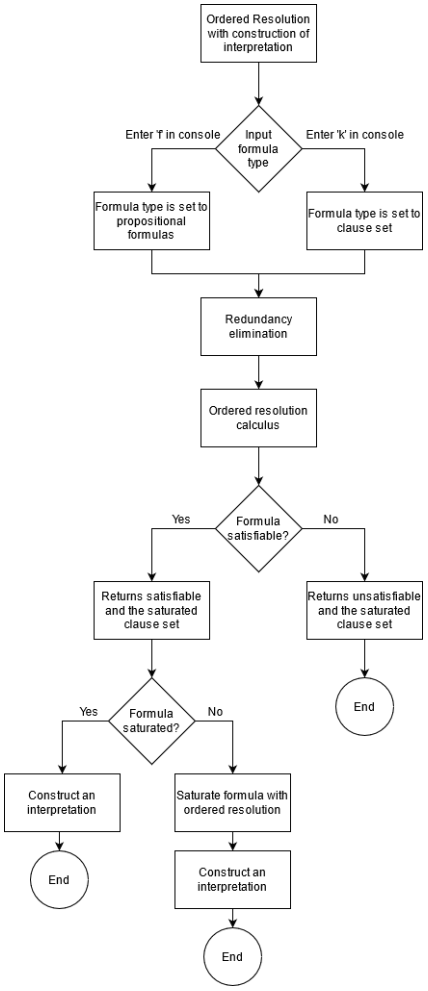



**Figure 3.2:** Workflow diagram of the ordered resolution test.

calculus receives a formula and an ordering using the method described in Section 3.2.2. In the beginning, redundant clauses of the clause set are eliminated. Redundancy elimination allows the calculations for the ordered resolution to be more efficient as there exist fewer possible clauses for the inferences. This is explained in Section 2.3.2.5 and how it is implemented is explained in Section 3.6.2. Now, the clause set without the redundant clauses is used in the ordered resolution calculus which is explained in Section 2.3. If the formula is unsatisfiable, the test terminates saying the clause set is unsatisfiable. If the formula is satisfiable, the user must check whether the clause set is saturated up to redundancy if he wants to construct an interpretation. A set $N$ of clauses is saturated up to redundancy w.r.t. the ordered resolution calculus $Res_{\succ}$ if all resolvents and factors of clauses in $N$ are in $N$ or are redundant w.r.t. $N$ (i.e. are entailed by clauses in $N$ which are strictly smaller w.r.t. $\succ$). If the formula for the construction of an interpretation is not saturated up to redundancy, at least one clause is not minimal, which means that

the result of the construction may not be a model of the clause set. The problem of the existence of a minimal counterexample is explained in the main ideas of the construction in Section 2.3.2.2. However, if the user confirms that the clause set is saturated, a model for the clause set is constructed, and a valuation for the clause set will be returned. After that the test terminates.

### 3.1.2 Workflow of the redundancy test

Here is the reference to the workflow diagram Figure 3.3. One can choose between

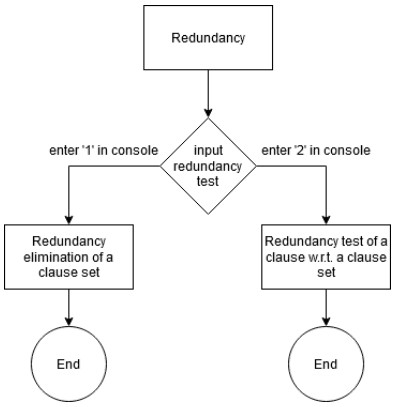



**Figure 3.3:** Workflow diagram of the redundancy test.

redundancy elimination of a clause set and redundancy test of a clause w.r.t. a clause set. The redundancy elimination requires a formula in clause set syntax as shown in Section 3.2.1.3, and an ordering in its syntax as presented in Section 3.2.2. All redundant clauses of the clause set are eliminated, and the output is the clause set without the redundant clauses, without changing its value. The implementation is presented in Section 3.6.2. Redundancy of a clause w.r.t. a clause set requires a formula in clause set syntax, an ordering, and a clause. This test tells if the clause is redundant w.r.t the entered clause set and the ordering. This will be further examined in Section 3.6.1.

### 3.1.3 Workflow of formula transformations

Four different transformations are possible as shown in Figure 3.4. Details about the implementation are presented in Section 3.4. The user has the option between four different transformations.

1. Transformation to NNF using equivalences,
2. Transformation to CNF using equivalences,
3. Transformation to DNF using equivalences, and
4. Transformation to CNF using structure preserving transformations.

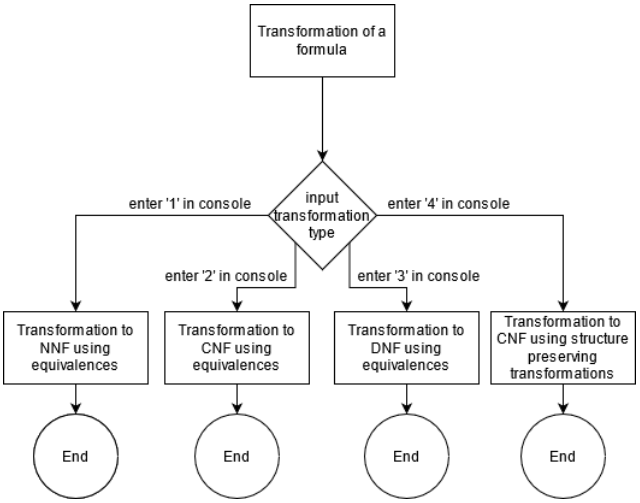



**Figure 3.4:** Workflow diagram of the transformation test.

All transformations require a propositional formula as an input. The first transformation, transformation to NNF using equivalences, transforms any propositional formula into an equivalent formula in NNF. It uses the first four of the six steps for the CNF transformation using equivalences in propositional logic in Section 2.2.1. No further explanations are presented in the teaching mode. The second transformation, transformation to CNF using equivalences, implements the six steps for the CNF transformation using equivalences in propositional logic, as presented in Section 2.2.1. This transformation returns an equivalent formula in CNF to the entered propositional formula. Transformation to CNF using equivalences does not differentiate between the teaching mode and the result mode. The output for these two modes is the same and no further explanations are presented. The third transformation, transformation to DNF using equivalences, transforms any propositional formula into an equivalent formula in DNF. No matter what mode this test is running in, the output stays the same with no further explanations. The fourth transformation, transformation to CNF using structure preserving transformations, expects a propositional formula in NNF for simplification purposes. However, one can also enter any propositional formula, but this formula will be automatically converted to an equivalent formula in NNF, and after that, the polarity based transformation operates as presented in Section 2.2.2 and defined in 2.18.

### 3.1.4 Workflow of the computing interpolant test

The general structure of the computing interpolant test can be found in Figure 3.5. In the computing interpolant test, we derive the interpolant of two entered clause sets according to the method for computing Craigs interpolant presented in Section 2.4. One chooses between an automatically generated ordering or manually constructed ordering. If it is manually created, then the user must enter the variables for the ordering according to Definition 2.39. Otherwise, the ordering is automatically created by extracting pro-

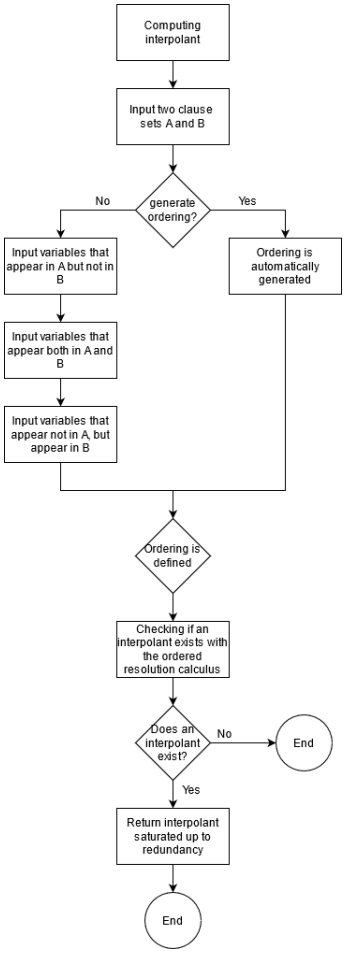
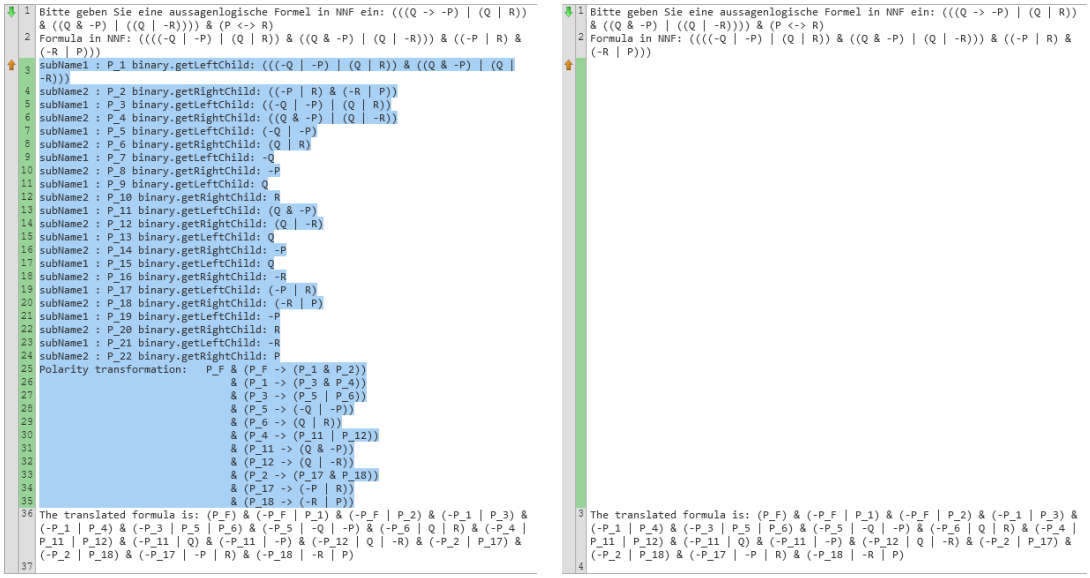



**Figure 3.5:** Workflow diagram of the computing interpolant test.

positional variables of two clause sets $A$ and $B$ and identifying the sets of propositional variables that occur only in specific formulas: $\Pi_A$ for $A$, $\Pi_B$ for $B$, $\Pi_{AB}$ for $\Pi_A \cap \Pi_B$. The atom ordering $\succ$ is defined, such that the propositional variables in $\Pi_A$ are strictly larger than those in $\Pi_{AB} \cup \Pi_B$. This procedure is described in Definition 2.39. After the ordering is defined, the test checks if an interpolant exists by applying the ordered resolution calculus on the union of the two entered clause sets. The test terminates if no interpolant exists, and the test returns an interpolant that is saturated up to redundancy if one exists. We will talk about the implementation in Section 3.7.

### 3.1.5 Teaching mode and result mode

We introduce two program modes, with two different purposes. The teaching mode's purpose is to print additional explanations of what the program is doing, to help understand how the presented aspects of propositional logic can be implemented and how the result is calculated. The result mode, however, prints only the necessary information on how the




**(a)** Structure preserving CNF transformation in teaching mode.

**(b)** Structure preserving CNF transformation in result mode.

**Figure 3.6:** Comparison of teaching and result mode on structure preserving CNF transformation.

result is calculated, and its purpose is to use the toolkit for work in propositional logic. The program mode is set before the test selection as can be seen in Figure 3.1. Depending on which test one is running, the test output in the two modes may have a big difference. For example in teaching mode, the conversion to CNF using structure-preserving transformation as explained in Section 3.4.4 additionally shows how each tree node of the formula is substituted and how the polarity transformation is obtained, whereas running the same test in result mode only returns the translated formula in CNF. The output is presented in Figure 3.6[6]. The following tests do not differ between the two modes: The conversion to CNF using equivalences, the conversion to DNF using equivalences, and the conversion to NNF using equivalences. The computation of an interpolant differs only insignificantly, which is not worth mentioning. The following differences are the most significant in the teaching mode: The ordered resolution calculus prints how inferences are made and how a model is constructed, the redundancy elimination shows which clause is redundant w.r.t. the corresponding part of the clause set, the structure preserving CNF transformation shows the substitution of the tree nodes as well as the polarity based transformations, and the polarity of a subformula test additionally prints the address of each subformula.

[6] `https://text-compare.com/`

## 3.2 Data structures

This section is about the data structures used in the toolkit for the implementation. Since we need our information organized for better algorithm efficiency, we have defined a data structure for the smallest units in our program. We implemented a data structure for literal, clause, and clause set, which are discussed in Section 3.2.1.3, a data structure for the ordering on clauses and clause sets, which are discussed in Section 3.2.2, and a data structure with an evaluator for the propositional formula, which is discussed in Section 3.2.3. Before we talk about the implemented data structures, we provide a proper definition for data structures in general.

**Definition 3.1 (Data structure [3]).** *A data structure is an organization of information, usually in memory, for better algorithm efficiency, such as queue, stack, linked list, heap, dictionary, and tree, or conceptual unity, such as the name and address of a person. It may include redundant information, such as the length of the list or the number of nodes in a subtree.*

Now that we know what a data structure is, let us have a look at an example of a data structure more closely.

**Example 3.2 (Data structure of a linked list [4]).** *A linked list is an implemented list, with each item having a link to the next item. The "LinkedList" consists of three attributes: an int attribute "size", a node pointer "first", and a node pointer "last". The size counts the number of nodes in the linked list. When constructing a linked list, it is initially empty, therefore "size" is initialized as zero. The first node pointer is needed to have a link to the first node of the linked list. If every node is linked, we can iterate over the linked list by starting at the first node and access the successors of every node in the linked list. Each node knows its previous node and its next node, as they are attributes of the node class. When the linked list is empty, the first node and the last node are null. The last node is a link to the last node. This is illustrated in Figure 3.7. One can add elements into a linked list by using the "add" method with an element e as the parameter of the "add" method. The type of the element must be of the same type as the linked list, or else one might get an error saying that the arguments mismatch. Adding an element e into a linked list is equivalent to applying the "linkLast" method with the element e as its parameter and returning true. When linking an element e to the last node of the linked list, a link is set from the last node of the linked list to the "new node". This new node has got the last node of the linked list as its predecessor, element e as its value, and null as its successor. This new node is set as the last node of the linked list. If, however, the linked list is empty, then the first node of the linked list is set as the new node. Else, the successor attribute of the previous last node is set as the new node. Finally, the size of the linked list and the structurally modified counter will be incremented by one as we added a node into the linked list. The java implementation of the LinkedList class looks like this:*

```
1  public class LinkedList<E>
2      extends AbstractSequentialList<E>
3      implements List<E>, Deque<E>, Cloneable, java.io.Serializable
4
5      transient int size = 0;
6
7      /**
8       * Pointer to first node.
9       */
10     transient Node<E> first;
11
12     /**
13      * Pointer to last node.
14      */
15     transient Node<E> last;
16
17     /**
18      * Constructs an empty list.
19      */
20     public LinkedList() {
21     }
22
23     /**
24      * Links e as last element.
25      */
26     void linkLast(E e) {
27         final Node<E> l = last;
28         final Node<E> newNode = new Node<>(l, e, null);
29         last = newNode;
30         if (l == null)
31             first = newNode;
32         else
33             l.next = newNode;
34         size++;
35         modCount++;
36     }
37
38     /**
39      * Appends the specified element to the end of this list.
40      *
41      * @param e element to be appended to this list
42      * @return {@code true}
43      * (as specified by {@link Collection#add})
44      */
45     public boolean add(E e) {
46         linkLast(e);
47         return true;
48     }
49
```

```
50
51      private static class Node<E> {
52          E item;
53          Node<E> next;
54          Node<E> prev;
55
56          Node(Node<E> prev, E element, Node<E> next) {
57              this.item = element;
58              this.next = next;
59              this.prev = prev;
60          }
61      }
```

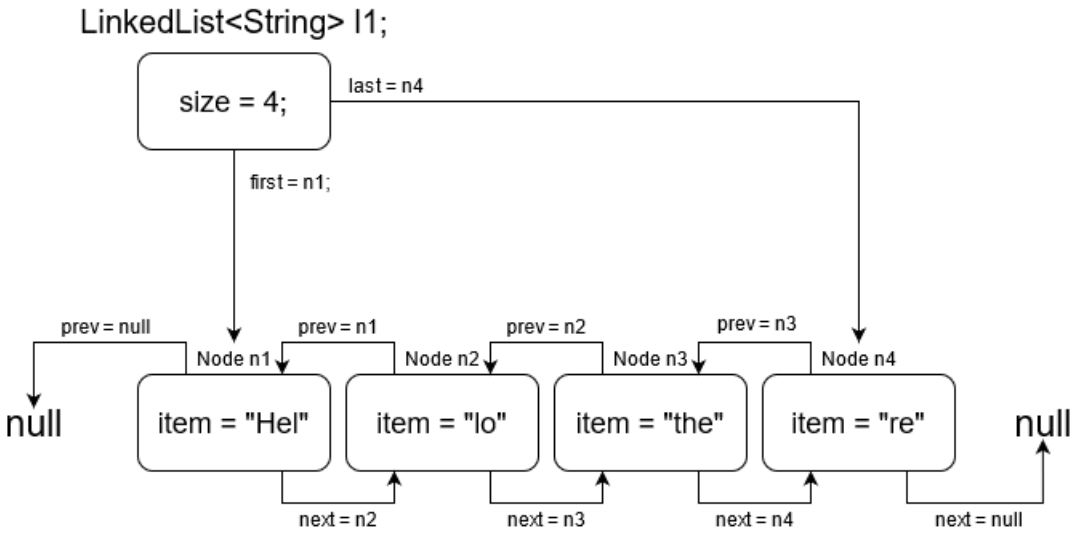



**Figure 3.7:** Illustration of a linked list.

The linked list is an important and common data structure that we also use for the clause and clause set, as we will see in the next section.

### 3.2.1 Data structures for literals, clauses, and clause sets

In this section, we present the implemented data structures for literals, clauses, and clause sets. We start with literal, continue with clause and clause set, and finalize with the syntax of a clause set for a test in the toolkit.

#### 3.2.1.1 Literal

A literal consists of two attributes: the propositional variable and the positive or negative value. For the propositional variable, we use a string attribute $s$, because we also want to subscript integers to the letters for more variety and a ”char” only allows a single

character. For the positive or negative value, we chose a boolean value, which is initially set for simplification purposes to true.

```
1  public class Literal {
2     /**
3       * Initially, set all the values
4       * of the literals to positive (true).
5       */
6      public boolean value = true;
7
8      /**
9       * Propositional variable.
10      */
11     public String s;
12
13     /***
14      * Constructor for a literal.
15      */
16     public Literal() {
17     }
```

#### 3.2.1.2 Clause

A clause is a multi-set of literals. There are different approaches for defining the data structure of a clause. One can choose a "bag" of literals or a "linked list" of literals. A bag is already a multi-set, which may contain duplicate elements. However, I prefer linked lists, because bags require an initial capacity, where linked lists do not. Linked lists are more dynamic as well. Swapping, adding, and removing elements at different places is easier. Also, the java library offers more predefined methods for linked lists. It is possible to use bags, but in this implementation, we have defined a clause as a linked list of literals.

```
1  public class Klausel implements Iterable<Literal>  {
2
3      /***
4       * Linked list of literals.
5       */
6      public LinkedList<Literal> l;
7
8      /***
9       * Constructor for a clause.
10      * @param l, linked list of literals.
11      */
12     public Klausel(LinkedList<Literal> l) {
13         this.l = l;
14     }
```

#### 3.2.1.3 Clause set

Since a clause set is a set of clauses, we implemented a clause set as a linked list of clauses. The constructor requires a linked list of clauses "n", and instantiates the internal attribute "n".

```
1  public class Klauselmenge implements Iterable<Klausel> {
2
3      /***
4       * A clause set is a linked list of clauses.
5       */
6      public LinkedList<Klausel> n;
7
8      /***
9       * Constructor for a clause set.
10      * @param n, linked list of clauses.
11      */
12     public Klauselmenge(LinkedList<Klausel> n){
13         this.n = n;
14     }
```

When entering a clause set for a test, the user must use the following notation.

**Syntax of a clause set for the toolkit**

- Positive literals consist of a propositional variable, which can be followed by an integer.

  For example the positive literal $A_1$ is represented as A1.

- Negative literals have a minus as a prefix.

  For example the negative literal $\neg A_3$ is represented as -A3.

- Clauses consist of literals, which are separated by a comma and placed in squared brackets.

  For example the clause $A_1 \vee \neg A_2 \vee A_3$ is represented as [A1,-A2,A3].

- Clause sets are clauses that are separated by a semicolon and placed in squared brackets.

  For instance the clause set $(A_1 \vee \neg A_2 \vee A_3) \wedge (\neg A_1 \vee A_2 \vee A_3) \wedge (\neg A_1 \vee \neg A_2)$ is represented as [[A1,-A2,A3];[-A1,A2,A3];[-A1,-A2]].

### 3.2.2 Ordering on clauses and clause sets

This section is about the ordering on literals. We start with the data structure of an ordering and how the syntax should be such that it can be entered as an input for a test in the toolkit. After that we examine the ordering functions for ordering a clause and ordering a clause set.

For the ordered resolution calculus we need an ordering on literals as described in Section 2.3. Clauses and clause sets are sorted according to the multi-set extension of that ordering on literals. We implemented the ordering on literals as a linked list of literals in descending order. The ordering defines the highest literal as the first node of the linked list, and the lowest literal as the last node of the linked list. When entering an ordering, the ordering must not have any duplicate literals, nor negative literals. Negative literals are automatically added according to their positive counterpart.

```
1  public class Ordnung
2  implements Comparable<Literal>, Iterable<Literal> {
3      /***
4       * The ordering is a linked list of literals,
5       * with some constraints.
6       */
7      public LinkedList<Literal> o;
8
9      /***
10      * Constructor for an ordering.
11      * Literals are sorted in a descending order.
12      *
13      * @param o, linked list of positive literals,
14      *     without duplicates.
15      */
16     public Ordnung(LinkedList<Literal> o){
17         if (enthaeltDuplikate(o))
18             throw new IllegalArgumentException("Ein Literal " +
19                     " kommt in der Ordnung doppelt vor");
20         this.o = o;
21         this.addNegativeLiterale();
22     }
```

To construct an ordering for a test, the user must use the following notation.

### Syntax of an ordering for the toolkit

- An ordering consists of positive literals, which are separated by a comma and placed in squared brackets.
- The order of the literals for the ordering are entered according to [A,B], if $A \succ B$.
- For the ordering, only positive literals must be entered.
- No duplicates must be entered.
- For example [A,P,B] represents the ordering $O := A \succ P \succ B$, which will be internally extended to [-A,A,-P,P,-B,B].

Now that we defined an ordering for the toolkit, we will talk about how clauses and clause sets are sorted according to an ordering. When sorting a clause with a given ordering,

we position the maximal literal as the first node of the linked list. The other literals are sorted according to that ordering. Let us have a look at Example 3.3 below.

**Example 3.3.** *JUnit test output for ordering a clause in clause notation with a given ordering. (Repetitions are allowed.)*
*Unsorted clause: [C,-C,A,R,B,B,-B,P,-P,-P,A].*
*Ordering: [-C,C,-R,R,-A,A,-P,P,-B,B].*
*Sorted clause: [-C,C,R,A,A,-P,-P,P,-B,B,B].*

Below one can see the implemented method for sorting a clause with a given order. The sorting algorithm used is "selectionsort" and operates in place.

```
1  public void ordnen(Klausel klausel){
2      testLiteraleInOrdnung(klausel);
3      for (int i = 0; i < klausel.size(); i++){
4           Literal kliteral1 = klausel.get(i);
5           for (int j = i + 1; j < klausel.size();j++){
6                Literal kliteral2 = klausel.get(j);
7                int position1 = o.indexOf(kliteral1);
8                int position2 = o.indexOf(kliteral2);
9                if (greaterThan(o.get(position2),o.get(position1))){
10                   Collections.swap(klausel.l,i,j);
11                   kliteral1 = klausel.get(i);
12                }
13           }
14      }
15  }
```

When sorting a clause set, every clause must be sorted. After that, we position the clause with the lowest maximal literal as the first node and the clause with the highest maximal literal as the last node of the linked list of clauses. The order is contrary to sorting a clause. If the maximal literals of two clauses are equal, then we compare their second-highest literal and so on, until the shorter clause will be positioned lower than the longer clause. One can see this in the output of an actual JUnit test of the toolkit in the Example 3.4 below.

**Example 3.4.** *JUnit test output for ordering a clause set in clause set notation with a given ordering.*
*Unsorted clause set: [[-A,A,P];[-A,A];[];[B,P,A];[B,B,-B,P,-P,-P,A];[-P,-A,-B]].*
*Ordering: [-A,A,-P,P,-B,B].*
*Sorted clause set: [[];[A,P,B];[A,-P,-P,P,-B,B,B];[-A,-P,-B];[-A,A];[-A,A,P]].*

The method for sorting a clause set is provided below and implements the sorting algorithm "insertionsort", which operates in-place.

```
1     public void ordnen(Klauselmenge klauselmenge){
2         for (Klausel klausel:klauselmenge) {
3             ordnen(klausel);
4         }
5
6
7         int n = klauselmenge.size();
8         for(int i = 1;i < n;i++){
9             int j = i - 1;
10            Klausel klausel1 = klauselmenge.get(j);
11            Klausel klausel2 = klauselmenge.get(j+1);
12            while(j >= 0 && kleinereKlausel(klausel2,klausel1)){
13                Collections.swap(klauselmenge.n,j,j+1);
14                j--;
15                klausel1 = klauselmenge.get(j);
16                klausel2 = klauselmenge.get(j+1);
17            }
18        }
19    }
```

### 3.2.3 Data structures for representing propositional formulas

The reason why we added the javaluator library into the toolkit is propositional formulas. Propositional formulas can be extended recursively, by substituting propositional variables of a formula with new formulas and placing them in brackets. Propositional formulas can be well expressed with a tree data structure. When adding the javaluator library, one can evaluate string expressions, by defining the operators of the respective language, its values, and its "evaluate" function in the *TreeEvaluator* class. When defining the operators, the first parameter is a string and it stands for the symbol of the operator. The second parameter stands for the number of arguments the operator receives. For example, one stands for a unary operator and two for a binary operator. The third argument defines the operator's associativity. For example the negation in $G \vee \neg F$ is right associative and applies to the right part of the formula $F$ instead of the $\vee$. The fourth argument sets the precedence of a formula similarly as already explained in Table 2.1. When our logical operators $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$, and parentheses are defined as parameters, we can define our values. Values are string expressions that are neither brackets nor operators. The defined values are "0" for falsum, "1" for verum, and string "s" for a new literal with "s" as its propositional variable. Whenever the *TreeEvaluator* evaluates an expression, it compares the string expression of that symbol with the symbols of the operators. If the evaluate function detects an operator, the operator is constructed in its respective class and its next arguments. The arguments can also be evaluated to other subformulas or values. This is how a tree data structure is constructed. If the evaluate function detects a symbol that is not defined, an exception is returned.

```
1 import com.fathzer.soft.javaluator.*;
2
3 import java.util.Iterator;
4
5 public class TreeEvaluator extends AbstractEvaluator<TreeNode> {
6
7     static final Operator NOT = new Operator("-", 1,
8 Operator.Associativity.RIGHT, 5);
9     static final Operator AND = new Operator("&", 2,
10 Operator.Associativity.LEFT, 4);
11     static final Operator OR = new Operator("|", 2,
12 Operator.Associativity.LEFT, 3);
13     static final Operator IMPLIES = new Operator("->", 2,
14 Operator.Associativity.LEFT, 2);
15     static final Operator EQUIV = new Operator("<->", 2,
16 Operator.Associativity.LEFT, 1);
17
18
19
20     private static final Parameters PARAMETERS;
21
22     static {
23         PARAMETERS = new Parameters();
24         PARAMETERS.add(NOT);
25         PARAMETERS.add(AND);
26         PARAMETERS.add(OR);
27         PARAMETERS.add(IMPLIES);
28         PARAMETERS.add(EQUIV);
29         PARAMETERS.addExpressionBracket(BracketPair.PARENTHESES);
30     }
31
32     public TreeEvaluator() {
33         super(PARAMETERS);
34     }
35
36     @Override
37     protected TreeNode toValue(String s, Object o) {
38         if (s.equals("0")){
39             return new Falsum();
40         }
41         if (s.equals("1")){
42             return new Verum();
43         }
44         else{
45             return new Literal(s);
46         }
47     }
48
49     @Override
```

```
50      public TreeNode evaluate(Operator operator,
51                               Iterator<TreeNode> operands,
52                               Object expr) {
53          if (operator == NOT){
54              return new Not(operands.next());
55          }
56          if (operator == AND){
57              return new And(operands.next(),operands.next());
58          }
59          if (operator == OR){
60              return new Or(operands.next(),operands.next());
61          }
62          if (operator == IMPLIES){
63              return new Implication(operands.next(),
64                                     operands.next());
65          }
66          if (operator == EQUIV){
67              return new Equivalence(operands.next(),
68                                     operands.next());
69          }
70          throw new IllegalArgumentException();
71      }
72  }
```

As we can see, the rules for entering a string expression must be strict to have a correct evaluation of our string into its representative propositional formula. Otherwise, we will get an exception. Therefore, we present the rules for entering a propositional formula as a string input.

### Syntax of a propositional formula

- Operators are entered according to the notation shown in Table 3.1.
- Propositional variables start with a letter and can be followed by an integer.
- Parentheses can be left out if an operator with higher precedence connects formulas with lower precedence.
- For example the entered string for a propositional formula
  (((Q -> -P) | -(Q | R)) & ((Q & -P) | ((Q | -R)))) & (P <-> R)
  represents the propositional formula
  $(((Q \rightarrow \neg P) \vee \neg(Q \vee R)) \wedge ((Q \wedge \neg P) \vee ((Q \vee \neg R)))) \wedge (P \leftrightarrow R)$.

**Table 3.1:** Symbols of operators for the toolkit.

| Symbol | Operator | Precendence |
|---|---|---|
| - | $\neg$ | 5 |
| & | $\wedge$ | 4 |
| \| | $\vee$ | 3 |
| -> | $\rightarrow$ | 2 |
| <-> | $\leftrightarrow$ | 1 |

## 3.3 Polarity of a subformula of a given formula

The polarity of a subformula of a given formula is a test that is provided in the toolkit. In Section 2.1 we provided Definition 2.3 for the position of a subformula in a given formula, and Definition 2.6 for polarities of a subformula. In this section, we will present the implementation of the polarity test. The polarity test takes a formula $f$ and a subformula $g$ of $f$ as a string input. Both formulas must be evaluated by the *TreeEvaluator* as presented in Section 3.2.3. The PolarityProcesser processes the evaluated tree nodes and returns the result of this method. The result is the polarity of the subformula with its address. The procedure is shown in Algorithm 1. The main part of the polarity

**Algorithm 1** Polarity of a subformula of a given formula

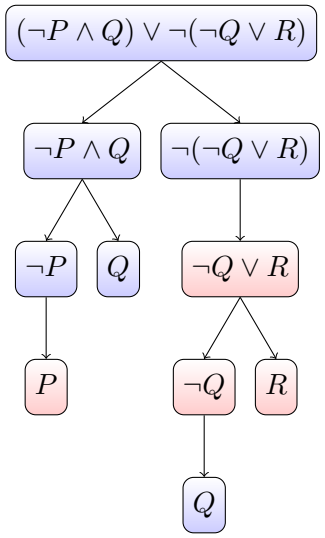
```
1: procedure POLARITY(f, g)          ▷ Formula f as string, subformula g as string
2:     treeF ← TreeEvaluator.evaluate(f)
3:     treeG ← TreeEvaluator.evaluate(g)
4:     result ← PolarityProcessor.process(treeF, treeG)
5:     return result                 ▷ Polarity of the subformula with its address
```

computation can be found in the process method of the PolarityProcessor class. The process function eliminates equivalences and implications of the formula according to the first and second step of the CNF transformation using equivalences as explained in Section 2.2.1. After that every node in the formula tree is processed, starting with the first tree node. The polarity processor counts the number of negations for every tree node. If the number of negations is even, the polarity of the node is positive. If the number of negations is odd, the polarity of the node is negative. One can see the procedure in Table 3.2.

For every tree node that matches the wanted subformula, the polarity and the position of the treenode is saved as described in Definition 2.3. When all treenodes are processed, every treenode that matches the wanted subformula is returned with its position and polarity.

**Table 3.2: Tree based polarity processor.**

Blue for positive polarity, red for negative polarity.




## 3.4 Conversion of formulae

This section covers the implementation of formula transformation tests. We will start with the NNF transformation because it is a part of the transformation of a formula into a formula in CNF using equivalences. Then we will show how marginally the CNF transformation using equivalences must be adjusted to compute an equivalent formula in disjunctive normal form for any given formula. Last but not least, we present a different approach to generate a formula in CNF if the formula in CNF only has to be equisatisfiable w.r.t. the entered formula.

### 3.4.1 Conversion to NNF using equivalences

When transforming a formula in NNF, one method is to perform Step 1 - 4 shown in Section 2.2.1 on our formula. Firstly, we need to evaluate the entered propositional formula. As explained in Section 3.2.3, the javaluator library provides a *TreeEvaluator* class with an evaluation function to construct a tree for our respective formula, which is entered as a string. The tree is processed by the conversionToNNF.process method, which applies Step 1 - 4 on the tree. When Step 1 - 4 is performed, we can return the resulting tree as a string. This general procedure is shown in Algorithm 2. Now we will have a closer look at how Step 1 - 4 is implemented. The first step is about eliminating equivalences, by splitting an equivalence into two implications. For example $F \leftrightarrow G$ is transformed into $(F \rightarrow G) \wedge (G \rightarrow F)$. Whenever the tree node with an instance of an equivalence operator is spotted, the tree node with the equivalence is replaced by a new tree node, which is a conjunction of two implications. The arguments of the equivalence are passed to the implications following the example above. The second step is about

**Algorithm 2** Conversion to NNF using equivalences

```
1: procedure CONVERSION TO NNF(f)                    ▷ Formula f as string
2:     tree ← TreeEvaluator.evaluate(f)
3:     tree ← conversionToNNF.process(tree)
4:     result ← tree.toString()
5:     return result                                 ▷ An equivalent formula in NNF

1: procedure CONVERSIONTONNF.PROCESS(tree)           ▷ Tree for the entered formula f
2:     tree ← firstStep.process(tree)
3:     tree ← secondStep.process(tree)
4:     tree ← thirdAndFourthStep.process(tree)
5:     return tree                                   ▷ An equivalent formula in NNF
```

replacing implications with disjunctions. Remember $F \to G$ is transformed into $\neg F \vee G$. Tree node instances with an implication are replaced by a new tree node, which is a disjunction of the negated first argument and the second argument of the implication. The third step pushes negations downward. Whenever a tree node with an instance of negation is spotted, we must consider several cases.

- If the child of the node is a value, then we have finished a tree branch and can return to the parent node. A value node can be $\bot$ which is represented as 0, $\top$ which is represented as 1, or a literal.
- If the child of the node is a conjunction, then we replace the current node with a disjunction of the negated arguments of the conjunction.
- If the child of the node is a disjunction, then we replace the current node with a conjunction of the negated arguments of the disjunction.
- If the child of the node is a negation, then we replace the current node and the child of the node with the child of the child of the current node. This means that we will just continue processing our tree two negations downward while ignoring two negations in a row.

When all these transformations are done, Step 1 - 4 are completed, which results in an equivalent formula in NNF.

### 3.4.2 Conversion to CNF using equivalences

For the conversion of any formula in an equivalent formula in CNF, we need to apply Step 1 - 6 on the entered formula, which is presented in Section 3.2.3. Again we have to use the *TreeEvaluator* to construct a tree for evaluating propositional formulas as a string. After that, we process the tree with the conversionToCNF.process method, which applies changes to the tree structure according to Step 1 - 6. The first, second, third, and fourth step are the same methods that are used for computing the NNF, so we can reuse these methods. However, step five and six are needed to transform any formula in

NNF to an equivalent formula in CNF. Let us have a look at the implementation of Step 5 and Step 6. Step 5 is about pushing disjunctions downward. For example $(F \wedge G) \vee H$

**Algorithm 3** Conversion to CNF using equivalences

```
1: procedure CONVERSION TO CNF(f)                          ▷ Formula f as string
2:     tree ← TreeEvaluator.evaluate(f)
3:     tree ← conversionToCNF.process(tree)
4:     result ← tree.toString()
5:     return result                                       ▷ An equivalent formula in CNF
1: procedure CONVERSIONTOCNF.PROCESS(tree)                 ▷ Tree for the entered formula f
2:     tree ← firstStep.process(tree)                      ▷ Also used in NNF
3:     tree ← secondStep.process(tree)                     ▷ Also used in NNF
4:     tree ← thirdAndFourthStep.process(tree)             ▷ Also used in NNF
5:     tree ← fifthStepCNF.process(tree)
6:     tree ← sixthStep.process(tree)
7:     return tree                                         ▷ An equivalent formula in CNF
```

is transformed to $(F \vee H) \wedge (G \vee H)$. By processing our tree we need to spot nodes that are instances of a disjunction and consider the following cases.

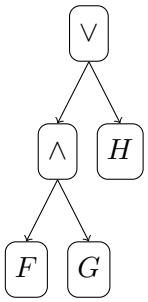
- The following case is represented in Table 3.3. If the left child of the disjunction is a conjunction, then we need to replace the disjunction with a new node, which is an instance of a conjunction, and make the following transformations on our tree.
  - We connect the left child of the conjunction with the right child of the disjunction by a new disjunction.
  - Then we connect the right child of the conjunction with the right child of the disjunction by a new disjunction.
- The following case is represented in Table 3.4. If the right child of the disjunction is a conjunction, then we need to replace the disjunction with a new node again, which is an instance of a conjunction, and make the following transformations:
  - We connect the left child of the conjunction with the left child of the disjunction by a new disjunction.
  - Then we connect the right child of the conjunction with the left child of the disjunction by a new disjunction.

The sixth Step eliminates $\top$ and $\bot$. When processing the tree, the following cases must be considered:

- If a node is an instance of a conjunction, and one child of the node is $\top$, we replace the node with the other child of the node.
- If a node is an instance of a conjunction, and one child of the node is $\bot$, we replace the node with $\bot$.

**Table 3.3: First case of Step 5 visualized.**

**(a)** Before transformation.




**(b)** After transformation.

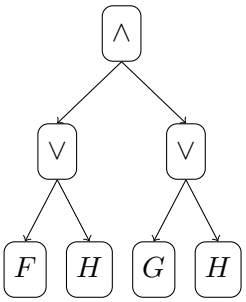
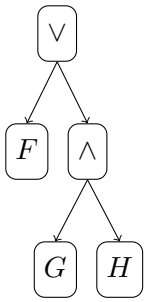
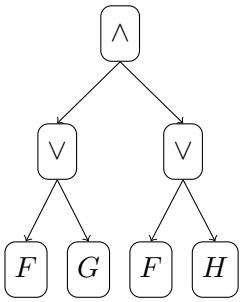



**Table 3.4: Second case of Step 5 visualized.**

**(a)** Before transformation.




**(b)** After transformation.




- If a node is an instance of a disjunction, and one child of the node is $\bot$, we replace the node with the other child of the node.
- If a node is an instance of a disjunction, and one child of the node is $\top$, we replace the node with $\top$.

When the sixth Step is complete, we can return the result, which is the transformed tree as a string.

### 3.4.3 Conversion to DNF using equivalences

As we will see, the conversion to DNF only differs marginally from the conversion to CNF in Section 3.4.2. Again, we need to use the *TreeEvaluator* to evaluate a formula entered as a string into a tree. This tree is passed to the conversionToDNF class to be processed. To transform any formula in DNF, one needs only to adjust step 5 from Section 2.2.1, the other steps are equivalent to the transformation of a formula into CNF. Instead of pushing disjunctions downward, we will push the conjunctions downward. This way our Step 5 for transforming a formula in DNF looks like this:

$$H[(F \vee F') \wedge G]_p \Rightarrow_{DNF} H[(F \wedge G) \vee (F' \wedge G)]_p$$

Now we need to adjust the method which is responsible for the fifth step to get a DNF instead of a CNF accordingly. When processing our tree we need to spot nodes which

**Algorithm 4** Conversion to DNF using equivalences

```
1: procedure CONVERSION TO CNF(f)                          ▷ Formula f as string
2:     tree ← TreeEvaluator.evaluate(f)
3:     tree ← conversionToDNF.process(tree)
4:     result ← tree.toString()
5:     return result                                       ▷ An equivalent formula in DNF
1: procedure CONVERSIONTODNF.PROCESS(tree)                 ▷ Tree for the entered formula f
2:     tree ← firstStep.process(tree)                      ▷ Also used in NNF/CNF
3:     tree ← secondStep.process(tree)                     ▷ Also used in NNF/CNF
4:     tree ← thirdAndFourthStep.process(tree)             ▷ Also used in NNF/CNF
5:     tree ← fifthStepDNF.process(tree)
6:     tree ← sixthStep.process(tree)                      ▷ Also used in CNF
7:     return tree                                         ▷ An equivalent formula in DNF
```

are an instance of a conjunction and consider the following cases.

- The following case is represented in Table 3.5. If the left child of the conjunction is a disjunction, then we need to replace the conjunction with a new node, which is an instance of a disjunction, and make the following transformations on our tree.
  - We connect the left child of the disjunction with the right child of the conjunction by a new conjunction.
  - Then we connect the right child of the disjunction with the right child of the conjunction by a new conjunction.
- The following case is represented in Table 3.6. If the right child of the conjunction is a disjunction, then we need to replace the conjunction with a new node again, which is an instance of a disjunction, and make the following transformations:
  - We connect the left child of the disjunction with the left child of the conjunction by a new conjunction.
  - Then we connect the right child of the disjunction with the left child of the conjunction by a new conjunction.

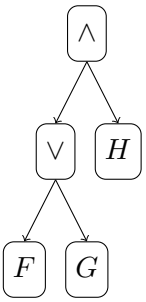
The transformation above is the only change we have to make to compute an equivalent formula in DNF instead of in CNF.

### 3.4.4 Conversion to CNF using structure preserving transformation

To compute a formula in CNF, we can weaken our constraints to apply a more efficient method. The structure preserving CNF transformations as presented in Section 2.2.2 return an equisatisfiable formula in CNF w.r.t the entered formula by computing the

**Table 3.5: First case of Step 5 in DNF visualized.**

**(a)** Before transformation.




**(b)** After transformation.

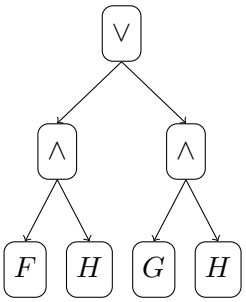
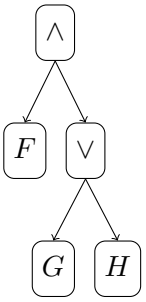
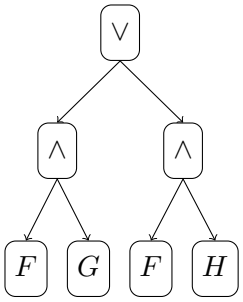



**Table 3.6: Second case of Step 5 in DNF visualized.**

**(a)** Before transformation.




**(b)** After transformation.




polarities of subformulae. However, we added another constraint to the method for simplification purposes. By constraining the user to enter a formula in NNF, we eliminate the possible occurrence of subformulae with a polarity of zero, and the occurrence of subformulae with a negative polarity, that are not literals. Therefore we need only the first rule of of the polarity based transformation, $(Q \to F)$ if $pol(H, p) = 1$, which is given in Section 2.18. If the user does not enter a formula in NNF, an equivalent formula in NNF is computed automatically, according to the computation of an equivalent formula in NNF as presented in Section 3.4.1. We assume that $f$ is the entered formula, for which we seek an equisatisfiable formula $g$ in CNF. Once again, we need the *TreeEvaluator* to construct a tree for our formula $f$. After that, we assure that the entered formula $f$ is in NNF by computing an equivalent formula in NNF for the entered formula. If the entered formula $f$ already is in NNF, then on the tree no further transformation are performed. Now we will explain the *StructurePreservingCNFProcessor* in more detail. The *StructurePreservingCNFProcessor* contains a string, which is initially empty and computed while processing the tree. Whenever a tree node is processed, a new name is invented for the subformula at the current tree node. This new name will be a new propositional variable that is used to substitute the subformula at the current node, visualized in Table 3.7. These new propositional variables are passed to a string builder, which connects the propositional variables according to the given definition $(Q \to F)$, if

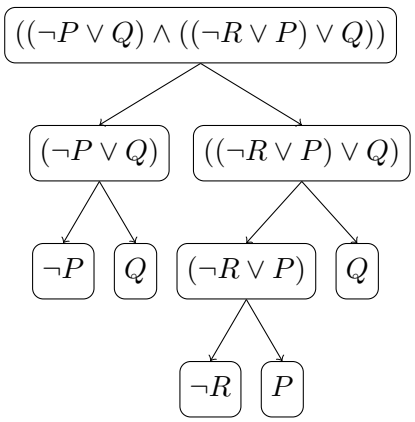
$pol(H, p) = 1$. This string is a new propositional formula $g$, which is equisatisfiable to $f$. However, the propositional formula $g$ is not in CNF, as it contains implications according to the definition $(Q \rightarrow F)$, if $pol(H, p) = 1$. We need to apply the second Step, the fifth Step, and the sixth step to transform $g$ into CNF. Finally, we can return the tree of g as a string, which is an equisatisfiable formula w.r.t. the entered formula $f$.

**Algorithm 5** Conversion to CNF using structure preserving transformations

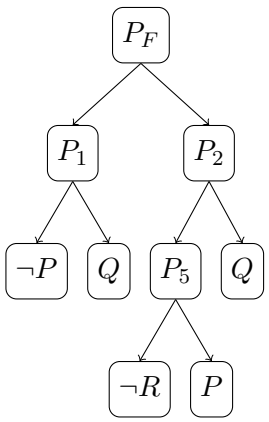
```
1:  procedure STRCT. PRES. TRANSFORMATION(f)        ▷ Formula f in NNF as string
2:      treeF ← TreeEvaluator.evaluate(f)
3:      treeF ← conversionToNNF.process(treeF)
4:      g ← strctPresCNF.process(treeF)             ▷ An equisatisfiable formula g
5:
6:      treeG ← TreeEvaluator.evaluate(g)
7:      treeG ← secondStep.process(treeG)           ▷ Also used in NNF/CNF/DNF
8:      treeG ← fifthStepCNF.process(treeG)         ▷ Also used in CNF
9:      treeG ← sixthStep.process(treeG)            ▷ Also used in CNF
10:     result ← treeG.toString()
11:     return result                               ▷ An equisatisfiable formula g in CNF
```

**Table 3.7: Visualization of subformulae substitution.**

**(a)** Before substitution.




**(b)** After substituion.



## 3.5 Ordered resolution calculus

In this section, we present the implementation of the ordered resolution calculus. The theoretical aspects of the ordered resolution calculus are explained in Section 2.3, so we will focus now on the implementation. As we will see, the ordered resolution calculus is a main part of the program. The calculus is used in several tests. Firstly, it is used to determine whether a clause set or a propositional formula is satisfiable, which will be discussed in this section. Secondly, it is used to eliminate redundant clauses of a clause set, which is discussed in Section 3.6. And thirdly, it is used to determine an interpolant for an unsatifiable union of two clause sets, which is discussed in Section 3.7. We start with the presentation and explanation of code extracts, which are used to apply the factorization rule and resolution rule in the ordered resolution calculus. We continue with the algorithm for the ordered resolution calculus, which uses the factorization and resolution rule, and determines whether a clause set is satisfiable. The construction of an interpretation finishes this section, and shows how a model is constructed for a satisfiable clause set in the toolkit.

### 3.5.1 Factorization rule

The purpose of the factorization rule is that whenever we have a clause that contains a literal that appears more than once, we can apply the factorization rule, and generate an equivalent clause that contains the duplicate literal only once. The formal definition can be found in Definition 2.25. In this implementation, we will not differentiate how often a duplicate literal appears in a clause. If the duplicate literal appears more than once, we reduce the occurrence to one. This approach differs slightly from the factorization rule in Definition 2.25, because the factorization rule reduces the occurrence $n \in \mathbb{N} : n > 1$ of a duplicate literal in a clause to $n - 1$. However, by applying the factorization rule $n - 1$ times, we would get the same result. Another important aspect is that in this implementation we will not differentiate whether the propositional variable is positive or negative. If it appears more than once, we will apply the factorization rule. This is worth mentioning because some definitions of the factorization rule are limited to the positive occurrence of duplicate literals. The negative factorization rule is given implicitly. In order to apply the factorization rule in our program, we must set a precondition. The first precondition is, that the clause on which the factorization rule is applied, must be sorted by an ordering on clauses. When iterating over the clause in the factorization method, we assume that the duplicate literals appear one after another. Duplicate literals will always appear one after another when we sort them by an ordering as defined in Section 3.2.2. By iterating over the clause, we simply eliminate literals that equal the previous literal in the clause. This method is in the complexity class $\mathcal{O}(n)$ if $n$ is the number of literals in the clause.

```java
    /***
     * Applies the factorization rule on a clause.
     *
     * Requirement: The clause must be ordered!
     * Iteration over the literals of an entered clause,
     * while deleting duplicates.
     *
     * Since the clause is sorted,
     * it is sufficient to iterate once over the clause.
     * The previous literal is compared with the current literal.
     * If the previous literal equals the current literal,
     * the current literal is deleted.
     * @param klausel, ordered clause that may contain duplicates.
     */

    public void faktorisierungsregel(@NotNull Klausel klausel) {
        Iterator<Literal> itr = klausel.iterator();

        if (itr.hasNext()) {
            Literal previous = itr.next();
            while (itr.hasNext()) {
                Literal current = itr.next();
                if (previous.equals(current)) {
                    itr.remove();
                } else {
                    previous = current;
                }
            }
        }
    }
```

### 3.5.2 Resolution rule

The definition of the ordered resolution calculus is already covered in 2.25. Now we will present the implementation of the resolution inference rule. In the toolkit, the ordered resolution inference rule can only be applied on two clauses if the following two conditions are met. Firstly, the propositional variable of the maximal literals of the clauses is equal, and secondly, the maximal literal of one clause is positive whereas the maximal literal of the other clause is negative. These conditions correspond to the definition given in Definition 2.25, and if they are not met for the entered clauses, an exception will be thrown. If the conditions are met for the entered clauses, a resolvent is returned. A resolvent is generated by creating a copy for both of the entered clauses, removing the maximal literal of both copies, and adding all the remaining literals of both copies into our new clause, which will be the new resolvent for the entered clauses. By creating copies for the entered clauses, we assure that we do not modify them. This assurance is needed, because the entered clauses will be coming from an existing clause set as we

will see in Section 3.5.3, and by modifying them, one could possibly change the valuation of the clause set. And by changing the valuation of a clause set, one can get a wrong answer according to the satisfiability of the clause set.

```
1  /***
2  * Applies the resolution rule on a clause.
3  *
4  * Requirements:
5  * The entered clauses are ordered.
6  * The maximal literals of the two entered clauses
7  * have the same propositional variable,
8  * but one is positive and the other negative.
9  * Otherwise, an exception is thrown.
10 *
11 * We delete the maximal literals of the clauses
12 * and merge them into a new clause.
13 *
14 * The entered clauses are not modified during the method.
15 *
16 * @param klausel1, first clause for the resolution rule.
17 *
18 * @param klausel2, second clause for the resolution rule.
19 *
20 * @return Klausel, resolvent clause.
21 */
22 public Klausel resolutionsregel(Klausel klausel1,
23                                 Klausel klausel2) {
24     Literal l1 = klausel1.l.peekFirst();
25     Literal l2 = klausel2.l.peekFirst();
26     assert l1 != null;
27     assert l2 != null;
28     if (l1.s.equals(l2.s) && l1.value != l2.value) {
29         Klausel klausel1Clone = new Klausel(new LinkedList<>());
30         klausel1Clone.addAll(klausel1);
31         Klausel klausel2Clone = new Klausel(new LinkedList<>());
32         klausel2Clone.addAll(klausel2);
33         klausel1Clone.l.pollFirst();
34         klausel2Clone.l.pollFirst();
35         Klausel merge = new Klausel(new LinkedList<>());
36         merge.addAll(klausel1Clone);
37         merge.addAll(klausel2Clone);
38         return merge;
39     }
40     throw new IllegalArgumentException();
41 }
```

### 3.5.3 Algorithm

In this section, we will present the algorithm that we have created for the ordered resolution calculus as discussed in Section 2.3. The algorithm embeds the functions for the factorization and resolution rule, which we have covered in Section 3.5.1 and 3.5.2. The ordered resolution algorithm requires an ordering on literals and a clause set as defined in Section 3.2.2 and Section 3.2.1.3. In the beginning, the clause set will be sorted according to the method which is presented in Section 3.2.2. In the outer "for" loop, we iterate over the clause set. Whenever we notice an empty clause by the if statement, we can tell that the clause set is unsatisfiable. An empty clause can be detected by checking the size of each clause in the outer "for" loop. If clause k1 has a size of zero, then clause k1 contains no literal, which means that we have derived an empty clause. By iterating over the clause set, we also apply the ordered factorization inference rule on every clause preventively, because we cannot tell if a clause contains duplicates or not. The factorization method is presented in Section 3.5.1. If the clause does not contain any duplicates, the factorization method will not delete any literal in the clause. After assuring that the first clause for the ordered resolution inference rule is not an empty clause, and does not contain any duplicates, we extract the maximal literal of the first clause. In the inner "for" loop, we are looking for a matching second clause, by checking if the propositional variable of the second clause is equal to the propositional variable of the first clause and if the maximal literal of the second clause is negative. Since the clause set is sorted from the lowest clause to the highest clause, and the negative literals are ranked higher than positive literals, we can tell that if the first clause for our ordered resolution inference rule has a negative maximal literal, no matching higher clause than the first clause can be found in the clause set. This is the reason why we have an if condition in the inner "for" loop, which breaks out of the inner "for" loop, as no matching second clause can be found for the first clause. This break statement allows us to release the current first clause, and declare the next clause in the outer "for" loop as the next first clause for a possible inference. If, however, a second clause matches the first clause, then we apply the ordered resolution inference rule on the first and second clause. The ordered resolution inference rule is presented in Section 3.5.2. The resolvent of the inference will be sorted and screened for duplicates by the factorization inference rule. When I thought about an algorithm for the ordered resolution calculus, I had the problem that one can apply the same resolution inference rule on two clauses indefinitely many times, since the used clauses for the inference rule are not deleted from the clause set. The output was an indefinitely long clause set, that contained the resolvent indefinitely many times. I had to add an if condition to avoid that. So now, only if the clause set does not contain the resolvent in the clause set, the resolvent will be added and sorted into the clause set. Otherwise, the resolvent will be dismissed, since it already appears more than once in the clause set and would be redundant. After the resolution inference rule, we will start the outer "for" loop with $i = 0$ once again, until an empty clause can be derived, which would mean that the clause set is unsatisfiable, or no new resolvent of any ordered resolution inference can be added into the clause set, which would mean that the clause set is satisfiable.

Further explanations that are printed by the method are left out in the code below for the sake of simplicity.

```
1  /***
2  * Applies the ordered resolution calculus on clause set
3  * with a given ordering.
4  * Returns whether a clause set is satisfiable or not.
5  *
6  *
7  * @param ordnung, ordering for the clauses.
8  * @param klauselmenge, clause set
9  * @return boolean, false if the clause set is unsatisfiable,
10 * else true
11 */
12     public boolean geordneteResolution(Ordnung ordnung,
13                           Klauselmenge klauselmenge) {
14
15         ordnung.ordnen(klauselmenge);
16
17         for (int i = 0; i < klauselmenge.size(); i++) {
18             Klausel k1 = klauselmenge.get(i);
19             if (k1.size() == 0) {
20                 return false;
21             }
22             faktorisierungsregel(k1);
23             Literal l1 = k1.get(0);
24             for (int j = i + 1; j < klauselmenge.size(); j++) {
25                 Klausel k2 = klauselmenge.get(j);
26                 Literal l2 = k2.get(0);
27                 if (!l1.s.equals(l2.s)
28                     || l1.value == Literal.NOT) {
29                     break;
30                 } else {
31                     if (l1.value == true && l2.value == false) {
32                         Klausel merge = resolutionsregel(k1, k2);
33                         ordnung.ordnen(merge);
34                         faktorisierungsregel(merge);
35                         if (!klauselmenge.n.contains(merge)) {
36                             klauselmenge.n.add(merge);
37                             ordnung.ordnen(klauselmenge);
38                             i = -1;
39                             break;
40                         }
41                     }
42                 }
43             }
44         }
45         return true;
46     }
```

### 3.5.4 Construction of a model

In this section, we will cover the implementation of how a model is constructed for a satisfiable clause set. The theoretical aspects of the construction of an interpretation are already covered in Section 2.3.2.2. The method to construct a model for a satisfiable clause set requires an ordering on literals and a clause set. We initialize a model, which contains all the negative literals that occur in the ordering. With this model, we will check whether a clause is true under the current set of literals. The model clause corresponds with $I_C$, which is the set of propositional variables that are true for the clause set. In the next three "for" loops, we will start iterating over each literal of each clause of the clause set and compare the literal of the clause with every literal in the model. If a clause of the clause set contains a literal that already exists in our model, then we can tell that the clause is true under the current model. We set the boolean variable *nextKlausel* as true, which indicates that we do not need to adapt our model and can move on to the next clause with the break statement. However, if a clause does not contain any literal that exists in the model, then we must set the maximal literal of the current clause in the model as true. This procedure is done when the boolean value *nextKlausel* is still false after comparing each literal of the current clause with every literal of the model. After finishing the iteration over the clause set, we will return a model for the clause. Further explanations that are printed by the method are left out in the code below for the sake of simplicity.

```
1     /***
2      * Applies the construction of an interpretation
3      * on a satisfiable clause with a given ordering.
4      *
5      *
6      *
7      * Requirements: Clause set is saturated and ordered.
8      * @param klauselmenge, satisfiable clause set.
9      * @param ordnung, ordering on literals.
10     * @return LinkedList, list which contains a model
11     *                     for the clause set.
12     */
13    public LinkedList<Literal> kanonischeModellgenerierung(
14            Klauselmenge klauselmenge, Ordnung ordnung) {
15
16        // Initializes model,
17        // where every literal is set negative.
18        ordnung.ordnen(klauselmenge);
19        LinkedList<Literal> model = new LinkedList<Literal>();
20        for (int i = 0; i < ordnung.size(); i++){
21            if (i % 2 == 0){
22                model.add(ordnung.get(i));
23            }
24        }
25        boolean nextKlausel;
26        // Iterating over the clause of a clause set.
27        for (int i = 0; i < klauselmenge.size(); i ++){
28            nextKlausel = false;
29            Klausel temp = klauselmenge.get(i);
30            // Iterating over the literals of a clause.
31            for (int j = 0; j < temp.size(); j++){
32                if (nextKlausel) {
33                    break;
34                }
35                Literal literal = temp.get(j);
36               // Iterating over literals of the current model.
37                for (int k = 0; k < model.size(); k++){
38                    Literal literal2 = model.get(k);
39                    if (literal.equals(literal2)){
40               //Print(Formel mit Klausel.. ist wahr in Modell..)
41                        nextKlausel = true;
42                        break;
43                    }
44                }
45            }
46               // Adapting model.
47            if (!nextKlausel){
48                Literal maxLiteral = new Literal();
49                maxLiteral.value = false;
```

```
50                  maxLiteral.s = temp.get(0).s;
51                  for (int k = 0; k < model.size(); k++){
52                      if (model.get(k).equals(maxLiteral)){
53                          model.get(k).value = true;
54                  //Print(Literal.. ist strikt max. in Klausel..
55                  //Literal |=1);
56                      }
57                  }
58              }
59          }
60          return model;
61      }
```

## 3.6 Redundancy

In this section, we will cover the possible redundancy functions, which the toolkit provides. The redundancy tests rely heavily on the ordered resolution calculus that is covered in Section 3.5. The theoretical aspects of redundancy are given in Section 2.3.2.5. Now we will use the ordered resolution calculus to prove that a clause $C$ is redundant w.r.t. a clause set $N$, by proving that there are clauses $C_1, ..., C_n \in N$ with $n \geq 0$ such that $C_i \prec C$ for all $i \in \{1, ..., n\}$ and $C_1, ..., C_n \models C$. Since the ordered resolution can only prove unsatisfiability, we will prove that $C$ is redundant by showing that if $C_1, ..., C_n$ are all clauses in $N$ and strictly smaller (w.r.t. $\succ$) than $C$, then $(C_1, ..., C_n) \wedge \neg C$ is unsatisfiable.

### 3.6.1 Redundancy of a clause w.r.t. a clause set

The following test requires a clause, a clause set, and an ordering and returns whether the clause is redundant w.r.t. the clause set and the ordering. First of all, we will create a copy, let's call it *klauselmengeKlon*, for the entered clause set, and also add the entered clause into *klauselmengeKlon*. *klauselmengeKlon* will be sorted by the given ordering. Now we know which clauses of the entered clause set are smaller than the entered clause because we can iterate over *klauselmengeKlon* and extract all clauses that appear before the entered clause. The extracted smaller clauses are put into a new clause set, let´s call it *kleinereKlauseln*. To prove that the smaller clauses of the clause set entail the clause, we will negate the entered clause, which becomes a set of literals, add them into *kleinereKlauseln*, and sort them into their correct positions. This corresponds to proving whether $(C_1, ..., C_n) \wedge \neg C$ is unsatisfiable by the ordered resolution calculus. If the ordered resolution calculus returns that *kleinereKlauseln* is unsatisfiable with the given ordering, then we return that the entered clause is redundant w.r.t. the entered clause set and the ordering. If the ordered resolution calculus returns that *kleinereKlauseln* is satisfiable with the given ordering, then we return that the entered clause is not redundant w.r.t. the entered clause set and the ordering.

```
1
2  /***
3  * Tests whether a clause is redundant
4  * w.r.t.\ a clause set, ordering.
5  *
6  *
7  *
8  * @param klausel, clause which is tested for redundancy.
9  * @param klauselmenge, clause set.
10 * @param ordnung, ordering.
11 *
12 * @return boolean,  true -> the entered clause is redundant,
13 *                   false -> the entered clause is not redundant.
14 */
15 public boolean redundanztest(
16    Klausel klausel, Klauselmenge klauselmenge, Ordnung ordnung) {
17
18    Klauselmenge klauselmengeKlon = new Klauselmenge(
19                                       new LinkedList<>());
20
21    klauselmengeKlon.merge(klauselmenge);
22    klauselmengeKlon.add(klausel);
23    ordnung.ordnen(klauselmengeKlon);
24
25    Klauselmenge kleinereKlauseln = new Klauselmenge(
26                                     new LinkedList<>());
27
28    for (int i = 0; i < klauselmengeKlon.size(); i++){
29       if (klauselmengeKlon.get(i) != klausel){
30          kleinereKlauseln.add(klauselmengeKlon.get(i));
31       }
32       else {
33          break;
34       }
35    }
36    kleinereKlauseln.merge(klausel.negate());
37    ordnung.ordnen(kleinereKlauseln);
38    if (!geordneteResolution(ordnung, kleinereKlauseln)){
39       System.out.println("Redundant");
40       return true;
41    else{
42       System.out.println("Not redundant");
43       kanonischeModellgenerierung(kleinereKlauseln, ordnung);
44       return false;
45    }
46 }
```

### 3.6.2 Redundancy elimination

The method for redundancy elimination requires a clause set and an ordering. Every clause that is redundant in the clause set will be eliminated. Firstly, it is checked whether the clause set contains only one clause. A clause set with only one clause cannot be implied by smaller clauses in the clause set. After that, the clause set is sorted according to the ordering. We also initialize a clause set called *kleinereKlauseln*, which contains all the clauses smaller than the clause which is tested for redundancy. The second clause in the clause set is the first clause which is tested for redundancy. It will be negated and added with all the smaller clauses that are contained in *kleinereKlauseln* into a clause set called *vereinigteKlauseln*. This is illustrated in Figure 3.8. Now we can use the ordered resolution calculus to check whether *vereinigteKlauseln* is unsatisfiable. If *vereinigteKlauseln* is unsatisfiable, then the current clause which is tested for redundancy is redundant and will be removed from the clause set. If *vereinigteKlauseln* is satisfiable, then the current clause which is tested for redundancy is not redundant. Then we will expand *kleinereKlauseln* by the next clause and check whether the clause after the next one, is implied by the smaller clauses.

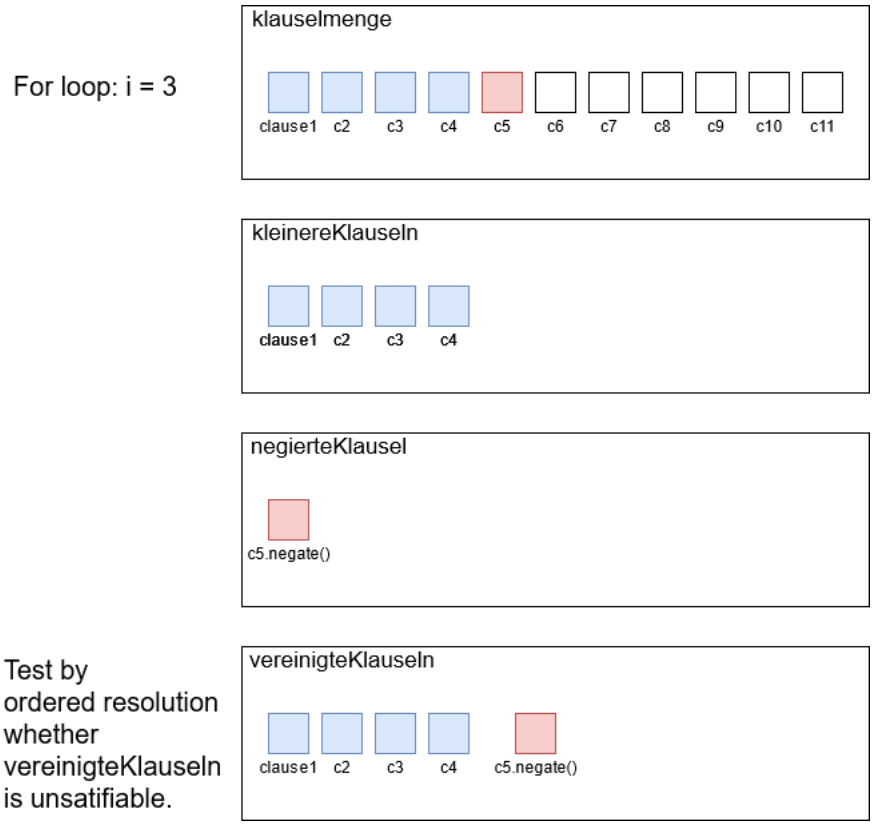



**Figure 3.8:** Illustration how redundant clauses are eliminated.

```java
/***
*  Deletes all redundant clauses in the clause set.
*
*  If vereinigteKlauseln is satisfiable,
*  then the current clause will not be eliminated.
*  If vereinigteKlauseln is unsatisfiable,
*  then the current clause will be eliminated in the clause set.
*
* @param klauselmenge, clause set for redundancy elim.
* @param ordnung, ordering on literals.
* @return Klauselmenge, clause set without redundant clauses.
*/

public Klauselmenge redundanzElimination(
   Klauselmenge klauselmenge, Ordnung ordnung){

   if (klauselmenge.size() <= 1){
      return klauselmenge;
   }
   ordnung.ordnen(klauselmenge);
   Klauselmenge kleinereKlauseln = new Klauselmenge(
                                       new LinkedList<>());
   for (int i = 0; i < klauselmenge.size(); i++){
      kleinereKlauseln.add(klauselmenge.get(i));
      if (i+1 < klauselmenge.size()){
         Klausel klausel = klauselmenge.get(i+1);
         Klauselmenge negierteKlausel = klausel.negate();

         Klauselmenge vereinigteKlauseln = new Klauselmenge(
                                              new LinkedList<>());
         vereinigteKlauseln.merge(kleinereKlauseln);
         vereinigteKlauseln.merge(negierteKlausel);

         if (!geordneteResolution(ordnung, vereinigteKlauseln)){
            System.out.println("Current clause redundant.");
            klauselmenge.n.remove(i+1);
            i--;
         }
         else {
            System.out.println("Current clause not redundant.");
         }
      }
   }
   return klauselmenge;
}
```

## 3.7 Interpolation

In this section, we present the implementation of the method for computing Craig-interpolants that is presented in Section 2.4. To calculate an interpolant, we require two clause sets $A$ and $B$, and multiple sets of propositional variables that occur only in specific formulas: $\Pi_A$ contains all the propositional variables that occur in $A$ but not in $B$, $\Pi_B$ contains all the propositional variables that occur in $B$ but not in $A$, $\Pi_{AB}$ contains all the propositional variables that occur in $A$ and in $B$, i.e. $\Pi_A \cap \Pi_B$. For the computation of an interpolant we receive the following arguments:

- Klauselmenge klauselmengeA, which corresponds to a clause set A.
- Klauselmenge klauselmengeB, which corresponds to a clause set B.
- Ordnung pureOrdnungA, which corresponds to $\Pi_A$.
- Ordnung ordnungCommon, which corresponds to $\Pi_{AB}$.
- Ordnung pureOrdnungB, which corresponds to $\Pi_B$.

The arguments *pureOrdnungA*, *ordnungCommon*, and *pureOrdnungB* can be entered manually or computed automatically. The function *pureOrdnung* compares every literal of the first clause set with every literal of the second clause set. If the propositional variable of a literal in the first clause set does not exist in the second clause set, then we add the propositional variable into an ordering which contains only literals that appear in the first but not in the second clause set.

```
1  public static Ordnung pureOrdnung(
2  Klauselmenge klauselmengeA, Klauselmenge klauselmengeB) {
3    Ordnung pureOrdnung = new Ordnung(new LinkedList<>());
4       for (int i = 0; i < klauselmengeA.size();i++){
5          for (int j = 0; j < klauselmengeA.get(i).size();j++){
6             boolean pure = true;
7             for (int k = 0; k < klauselmengeB.size();k++){
8               for (int l = 0; l < klauselmengeB.get(k).size();l++){
9                  String l1 = klauselmengeA.get(i).get(j).s;
10                 String l2 = klauselmengeB.get(k).get(l).s;
11                 if (l1.equals(l2)){ pure = false;}
12            }
13            }
14            if (pure){
15               pureOrdnung.add(new Literal (
16                           true, klauselmengeA.get(i).get(j).s));
17            }
18         }
19      }
20   return pureOrdnung;
21 }
```

The function *commonOrdnung* operates similarly. Every literal of the first clause set is compared with every literal of the second clause set. If the propositional variable of a literal in the first clause set does exist in the second clause set, then we add the propositional variable into an ordering that contains only literals that appear in both clause sets.

```
1  public static Ordnung commonOrdnung(
2  Klauselmenge klauselmengeA, Klauselmenge klauselmengeB) {
3
4    Ordnung commonOrdnung = new Ordnung(new LinkedList<>());
5    for (int i = 0; i < klauselmengeA.size();i++){
6        for (int j = 0; j < klauselmengeA.get(i).size();j++){
7            boolean common = false;
8            for (int k = 0; k < klauselmengeB.size();k++){
9              for (int l = 0; l < klauselmengeB.get(k).size();l++){
10                 String l1 = klauselmengeA.get(i).get(j).s;
11                 String l2 = klauselmengeB.get(k).get(l).s;
12                 if (l1.equals(l2)){
13                     common = true;
14                 }
15             }
16           }
17           if (common){
18               commonOrdnung.add(new Literal (
19                   true, klauselmengeA.get(i).get(j).s));
20           }
21       }
22   }
23   return commonOrdnung;
24 }
```

Since we have computed *pureOrdnungA*, *ordnungCommon*, and *pureOrdnungB*, we can call the interpolant function. Firstly, we have to check if an interpolant exists. If the union of *klauselmegeA* and *klauselmengeB* is unsatisfiable, then an interpolant exists. We merge the clause sets *klauselmengeA* and *klauselmengeB* into a new clause set called *vereinigungAB*, and generate an ordering for *vereinigungAB* called *ordnungAB*. The ordering consists of the literals that appear only in *klauselmengeA*, the literals that appear in *klauselmengeA* and *klauselmengeB*, and the literals that appear only in *klauselmengeB*. We use the ordered resolution calculus to determine whether the clause set *vereinigungAB* with the ordering *ordnungAB* is unsatisfiable. If it is satisfiable, then an interpolant does not exist and the test terminates. If it is unsatisfiable, we proceed with the computation of an interpolant. To get an interpolant, we need to saturate *klauselmengeA*, and extract the clauses from *klauselmengeA*, whose literals appear in *klauselmengeA* and *klauselmengeB*. We put the extracted clauses into a clause set called interpolante, which is the interpolant for the clause sets *klauselmengeA* and *klauselmengeB*. Finally, we eliminate redundant clauses in the interpolant and return the interpolant to the user.

```
1  public void interpolant(
2        Klauselmenge klauselmengeA,
3        Klauselmenge klauselmengeB,
4        Ordnung pureOrdnungA,
5        Ordnung ordnungCommon,
6        Ordnung pureOrdnungB) {
7
8     Ordnung ordnungAB = new Ordnung(new LinkedList<>());
9     ordnungAB.o.addAll(pureOrdnungA.o);
10    ordnungAB.o.addAll(ordnungCommon.o);
11    ordnungAB.o.addAll(pureOrdnungB.o);
12
13    Ordnung ordnungA = new Ordnung(new LinkedList<>());
14    ordnungA.o.addAll(pureOrdnungA.o);
15    ordnungA.o.addAll(ordnungCommon.o);
16
17    Klauselmenge vereinigungAB = new Klauselmenge(
18                                    new LinkedList<>());
19    vereinigungAB.addAll(klauselmengeA);
20    vereinigungAB.addAll(klauselmengeB);
21    // An interpolant exists if,
22    // and only if vereinigungAB is unsatisfiable.
23    if (!geordneteResolution(ordnungAB, vereinigungAB)) {
24       // Saturate klauselmengeA.
25       geordneteResolution(ordnungA, klauselmengeA);
26       // The interpolant consists of the clauses
27       // of the saturated clause set A,
28       // where all the literals are defined in the ordnungCommon.
29       // filterGemeinsameKlauseln extracts
30       // all the clauses from klauselmengeA,
31       // whose literals are all defined in ordnungCommon.
32       Klauselmenge interpolante = filterGemeinsameKlauseln(
33                                        ordnungCommon,
34                                        klauselmengeA);
35       // Eliminate redundant clauses in the interpolant.
36       redundanzElimination(interpolante, ordnungCommon);
37       System.out.println("Das ist die Interpolante: I = "
38                                        + interpolante);
39
40       } else {
41          System.out.println("ERROR:
42                              Es existiert keine Interpolante.\n");
43       }
44 }
```

# 4 Conclusion

This thesis is about the development of a toolkit in propositional logic, that relies on the ordered resolution calculus. Although there are different methods to prove satisfiability, like the semantic tableaux or the DPLL procedure, we chose the ordered resolution calculus for the following reasons. This toolkit provides tests that depend on the ordered resolution calculus like for example the canonical construction of a model, the redundancy tests, and the computation of an interpolant. As mentioned before, this toolkit should also serve didactical purposes, so being consistent with one procedure and showing its application in different tests is necessary. The easiest part of this work was to create and implement the algorithms for the ordered resolution calculus, redundancy tests, canonical construction, and the computation of an interpolant. The creation and implementation of these algorithms were exciting. A rather difficult part for me was the transformation of formulae. The transformation of a formula into an equivalent formula in CNF and the structure-preserving CNF transformations required a lot of time because I was not familiar with recursive operations on tree nodes. Fortunately, I could ask Dennis Peuter for guidance, since he had experience with javaluator. I also had to get used to scientific writing because one needs to have enormous attention to detail to make definitions and examples clear and uncomplicated. Thankfully, my supervisors Prof. Sofronie-Stokkermans and Dennis Peuter gave significant feedback, which helped me adapt to scientific writing. I hope that the toolkit finds application for people working with propositional logic, and the ordered resolution calculus. It can serve as a calculator and provide a variety of tests for satisfiability testing. But the greatest feature that sets the program apart from all the others, are the explanations of how the result is calculated. The transparency of calculations will make computations easily traceable for beginners and intermediates in propositional logic. Since there is a lot of possible future work, feedback and suggestions for improvement are always welcome.

## 4.1 Future Work

There are many possible extensions for the toolkit. Beginning with the ordered resolution calculus, it can be extended with a selection function. The algorithm can benefit from the selection function by smaller search spaces than with the unrestricted ordered resolution calculus. The user could then select negative literals and add them to a list. When performing the ordered resolution calculus with a selection function, we must add preconditions to apply the ordered resolution rule with selection and the ordered factorization rule with selection. Once we added them, the user could choose between the ordered resolution calculus and the ordered resolution calculus with a selection function.

Another extension opportunity is to implement a different method to compute the interpolant. In this toolkit, an interpolant is computed by saturating the first clause set and extracting clauses which contain only propositional variables that occur in both clause sets. However, one could implement a different method, which allows only inferences in the first clause set, with the highest propositional variable according to an ordering. It would be interesting to compare both methods and their resulting interpolants. Since the ordered resolution calculus is implemented only for propositional logic, one could extend it for first-order logic. Although the data structure must be adapted for expressions in first-order logic, there are a lot of applications for satisfiability testing in first-order logic, in program verification, and local graph theory, which would highly benefit from this toolkit. However, my favorite extension would be to implement a graphical user interface for the toolkit. The toolkit works fine as a command-line application, but a graphical user interface can be more appealing, and more user-friendly. I think a graphical user interface will benefit the toolkit the most. The toolkit can be embedded with a graphical interface as a Windows application, or into a website, like the existing SPASS[1] application, or even as an android application, depending on the purpose of the toolkit. There are various fascinating future projects which can result from the toolkit.

[1] `https://www.mpi-inf.mpg.de/departments/automation-of-logic/software/spass-workbench`

# Bibliography


[1] Franz Baader and Tobias Nipkow: *Term rewriting and all that.* Cambridge University Press, 1998, ISBN 978-0-521-45520-6.

[2] Leo Bachmair and Harald Ganzinger: *Completion of First-Order Clauses with Equality by Strict Superposition (Extended Abstract).* In Stéphane Kaplan and Mitsuhiro Okada (editors): *Conditional and Typed Rewriting Systems, 2nd International CTRS Workshop, Montreal, Canada, June 11-14, 1990, Proceedings*, volume 516 of *Lecture Notes in Computer Science*, pages 162–180. Springer, 1990. https://doi.org/10.1007/3-540-54317-1_89.

[3] Paul E. Black: *"data structure"*. in Dictionary of Algorithms and Data Structures [online], Vreda Pieterse and Paul E. Black, ed., 15 December 2004 2004. Available from: http://www.nist.gov/dads/HTML/datastructur.html (accessed 01 March 2021).

[4] Paul E. Black: *"linked list"*. in Dictionary of Algorithms and Data Structures [online], Vreda Pieterse and Paul E. Black, ed., 3 March 2020 2004. Available from: http://www.nist.gov/dads/HTML/datastructur.html (accessed 01 March 2021).

[5] William Craig: *Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory.* J. Symb. Log., 22(3):269–285, 1957. https://doi.org/10.2307/2963594.

[6] James Hein: *Discrete mathematics.* Jones and Bartlett Publishers, Boston, 2003, ISBN 0763722103.

[7] Kenneth L. McMillan: *Lazy Annotation for Program Testing and Verification.* In Tayssir Touili, Byron Cook, and Paul Jackson (editors): *Computer Aided Verification*, pages 104–118, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg, ISBN 978-3-642-14295-6.

[8] Jussi Rintanen: *Propositional Logic and Its Applications in Artificial Intelligence.* (1), 2017. Available from: https://mycourses.aalto.fi/pluginfile.php/273655/course/section/60890/notes-logic.pdf (accessed 11 March 2021).

[9] Viorica Sofronie-Stokkermans: *Propositional logic 1, CNF and ordered Resolution.* (2), 2019. Available from: https://userpages.uni-koblenz.de/ sofronie/lecture-dp-ws-2019/slides/prop-logic1.pdf (accessed 25 March 2021).

[10] Uwe Waldmann: *Automated Reasoning*. (1), 2019/2020. Available from: http://rg1-teaching.mpi-inf.mpg.de/autrea-ws19/script.pdf (accessed 25 March 2021).